

# Introduzione alla progettazione agile nello sviluppo software

**Sandro Pedrazzini**

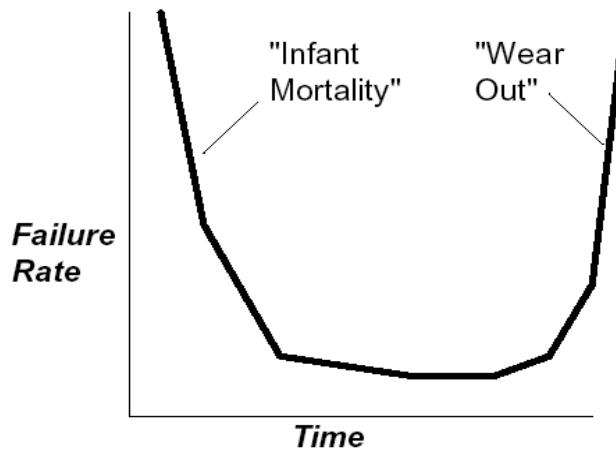
DTI-SUPSI  
[sandro.pedrazzini@supsi.ch](mailto:sandro.pedrazzini@supsi.ch)

Canoo Engineering AG  
[sandro.pedrazzini@canoo.com](mailto:sandro.pedrazzini@canoo.com)

## Contenuto

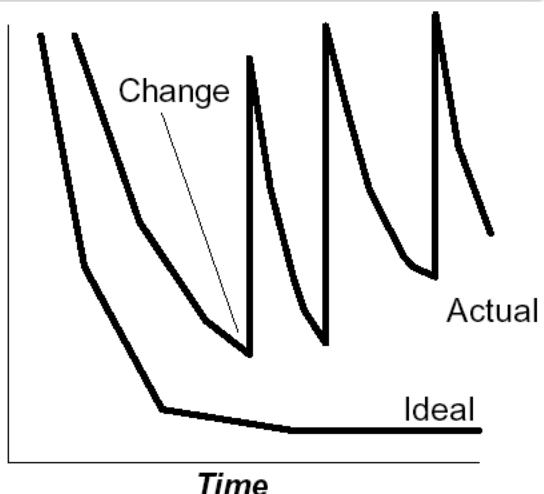
- **HW e SW**
- **Significato di progettazione agile**
- **Caratteristiche**
- **Manifesto agile**
- **Punti di forza**
- **Problemi**
- **Gestione del cliente**
- **Ruolo dello sviluppatore**

## Errori Hardware e Software (1)



**FAILURE CURVE  
FOR HARDWARE**

## Errori Hardware e Software (2)



**FAILURE CURVE  
FOR SOFTWARE**

Significato		SUPSI
	<b>Progettazione agile</b>	
	<ul style="list-style-type: none"> <li>• <b>Approccio diverso nella gestione di team e progetti in ambito sviluppo software</b></li> <li>• <b>“Formalizzata” nel 2001 con il manifesto agile, che propone una serie di valori da considerare</b></li> </ul>	

Sandro Pedrazzini

Progettazione agile nello sviluppo SW 5

Significato		SUPSI
	<b>Motivazione</b>	
	<ul style="list-style-type: none"> <li>• <b>Numero elevato di progetti interrotti</b></li> <li>• <b>Costi di pianificazione iniziale sproporzionati</b></li> <li>• <b>Non rispetto dei termini di consegna (e dei costi) nei progetti</b></li> <li>• <b>Rischio elevato per progetti di grosse dimensioni</b></li> </ul>	

Sandro Pedrazzini

Progettazione agile nello sviluppo SW 6

Significato	Motivazione (2)	SUPSI
	<ul style="list-style-type: none"> <li>• <b>Difficoltà di manutenzione</b></li> <li>• <b>Problemi di qualità</b></li> <li>• <b>Difficoltà nel coinvolgere il committente durante la fase di pianificazione</b></li> <li>• <b>Le aspettative dei committenti non vengono soddisfatte</b></li> </ul>	

---

Sandro Pedrazzini

Progettazione agile nello sviluppo SW 7

Significato	Motivazione (3)	SUPSI
	<ul style="list-style-type: none"> <li>• <b>Il contesto di un problema si modifica e si sviluppa più velocemente dei sistemi software</b></li> <li>• <b>Natura del software</b></li> <li>• <b>Miglior preparazione di chi si occupa di sviluppo</b></li> <li>• <b>Ruolo dello sviluppatore rivalutato</b></li> </ul>	

---

Sandro Pedrazzini

Progettazione agile nello sviluppo SW 8

Significato	<b>Fattori di differenziazione</b>	SUPSI
	<ul style="list-style-type: none"> <li>I costi della fase realizzativa, nei confronti di quelli della fase di pianificazione, hanno proporzioni diverse rispetto ad altri ambiti dell'ingegneria           </li> </ul>	
	 – Pianificazione   – Realizzazione 	
Sandro Pedrazzini		Progettazione agile nello sviluppo SW 9

Significato	<b>Fattori di differenziazione (2)</b>	SUPSI
	<ul style="list-style-type: none"> <li>L'attività di progettazione si interseca con quella realizzativa e ne trae spunto           </li> </ul>	
	<ul style="list-style-type: none"> <li>– Si fa del design anche durante lo sviluppo</li> <li>– La soluzione di problemi realizzativi può portare a modifiche di design</li> </ul>	
	 Con le metodologie agili non si riduce la fase di pianificazione, la si distribuisce diversamente all'interno del progetto	
Sandro Pedrazzini		Progettazione agile nello sviluppo SW 10

Significato	SUPSI
<b>Fattori di differenziazione (3)</b>	
<ul style="list-style-type: none"> <li>• <b>La percezione del cliente nei confronti del prodotto evolve nel tempo, man mano che il prodotto cresce in funzionalità</b> <ul style="list-style-type: none"> <li>– Migliore comprensione della soluzione che si desidera realizzare</li> <li>– Feedback di utenti di prova</li> <li>– Nuovi requisiti, non necessariamente marginali</li> </ul> </li> </ul>	

---

Sandro Pedrazzini
Progettazione agile nello sviluppo SW 11

Significato	SUPSI
<b>Fattori di differenziazione (4)</b>	
<ul style="list-style-type: none"> <li>• <b>La percezione del team di sviluppo nei confronti del prodotto evolve nel tempo, man mano che l'architettura del prodotto evolve</b> <ul style="list-style-type: none"> <li>– Migliore comprensione delle necessità del cliente</li> <li>– Visione più ampia del dominio di applicazione e della tecnologia</li> </ul> </li> </ul>	

---

Sandro Pedrazzini
Progettazione agile nello sviluppo SW 12

Caratteristiche		SUPSI
	<b>Caratteristiche “agili”</b>	
<ul style="list-style-type: none"> <li>• Necessità di coinvolgere il cliente in modo importante durante il processo di sviluppo</li> <li>• Il team di sviluppo è in grado di prendere decisioni</li> <li>• I requisiti del prodotto cambiano, ma il tempo complessivo di sviluppo rimane invariato</li> </ul>		
<hr/> <p>Sandro Pedrazzini</p> <p>Progettazione agile nello sviluppo SW 13</p>		

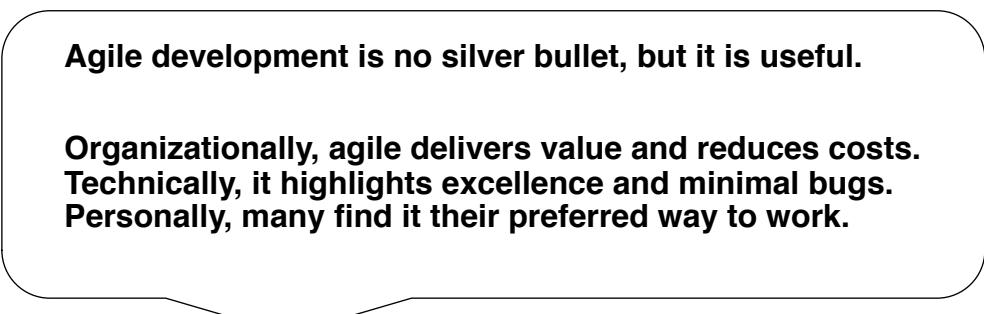
Caratteristiche		SUPSI
	<b>Caratteristiche “agili” (2)</b>	
<ul style="list-style-type: none"> <li>• Sviluppare piccole parti di prodotto alla volta, in brevi iterazioni</li> <li>• Concentrarsi sul rilascio frequente di nuove versioni del prodotto</li> <li>• Concentrarsi sulle richieste con maggiore priorità</li> </ul>		
<hr/> <p>Sandro Pedrazzini</p> <p>Progettazione agile nello sviluppo SW 14</p>		

Caratteristiche	Caratteristiche “agili” (3)	SUPSI
	<ul style="list-style-type: none"> <li>• <b>Integrare il test come elemento di sviluppo</b></li> <li>• <b>Mantenere un approccio collaborativo, sia all'interno del team, sia nei confronti degli stakeholders</b></li> <li>• <b>Approccio diverso nei confronti delle richieste di cambiamento di requisiti</b></li> </ul>	

---

Sandro Pedrazzini

Progettazione agile nello sviluppo SW 15

Caratteristiche	Perché Agile?	SUPSI
	<p><b>Agile development is no silver bullet, but it is useful.</b></p> <p><b>Organizationally, agile delivers value and reduces costs. Technically, it highlights excellence and minimal bugs. Personally, many find it their preferred way to work.</b></p>  <p>James Shore, The Art of Agile Development</p>	

---

Sandro Pedrazzini

Progettazione agile nello sviluppo SW 16

Caratteristiche	<b>Gestione agile vs. PM “classico”</b>	SUPSI
	<ul style="list-style-type: none"> <li>• <b>PM:</b> <b>necessaria una raccolta esaustiva dei requisiti</b></li> <li>• <b>Agile:</b> <b>i requisiti cambiano nel tempo e ne arrivano di nuovi in fase di sviluppo</b></li> </ul>	

---

Sandro Pedrazzini

Progettazione agile nello sviluppo SW 17

Caratteristiche	<b>Gestione agile vs. PM (2)</b>	SUPSI
	<ul style="list-style-type: none"> <li>• <b>PM:</b> <b>andando a fondo nei requisiti, la lista di task aumenta</b></li> <li>• <b>Agile:</b> <b>giusto, questo a maggior ragione quando si passa alla realizzazione</b></li> </ul>	

---

Sandro Pedrazzini

Progettazione agile nello sviluppo SW 18

### Gestione agile vs. PM (3)

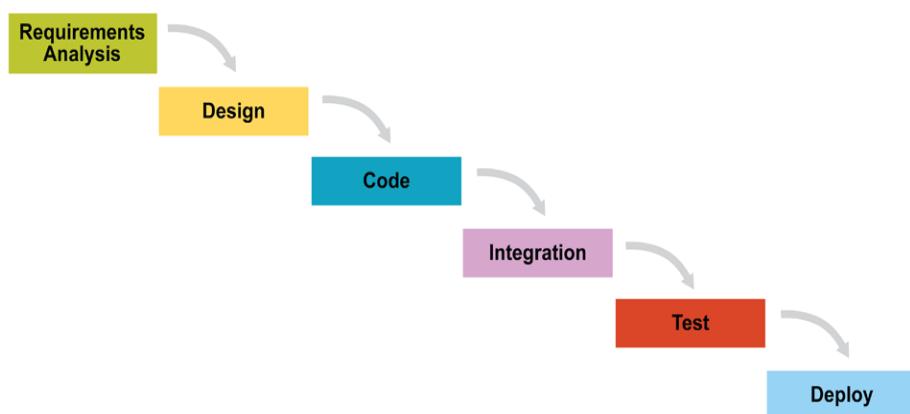
- **PM:**

**il progetto ha successo quando è riuscito a rispettare tempi, costi e requisiti**

- **Agile:**

**il progetto ha successo quando, nei tempi e costi prestabiliti, produce il software che risolve i problemi attuali del committente (business value)**

### Modello a cascata



## Caratteristiche

### Modello a cascata (2)

- **Caratterizzato dalla pianificazione e specificazione del sistema in ogni singolo dettaglio, prima di passare alla fase implementativa**
- **L'approccio a cascata è un modo rischioso e costoso per sviluppare sistemi software**
- **Per la realizzazione di applicazioni in cui i requisiti cambiano rapidamente durante il processo di sviluppo, le metodologie agili sono più adatte.**

Caratteristiche

SUPSI

## Modello a cascata (4)

- Studio sul fallimento di progetti informatici in UK, inizio anni 2000: nell'82% dei casi la gestione “waterfall-style” è stata giudicata come “il fattore singolo più determinante di insuccesso”
  - Craig Larson, Agile and Interactive Development

Sandro Pedrazzini

Progettazione agile nello sviluppo SW 22

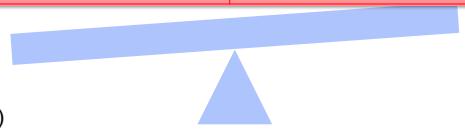
Caratteristiche	Sviluppo agile	SUPSI
	<p><b>“It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change.”</b></p> <p>– Charles Darwin</p>	

Caratteristiche	Cascata vs. sviluppo agile	SUPSI
	<ul style="list-style-type: none"> <li>• Le metodologie agili si basano sull’approccio incrementale alla specifica e allo sviluppo software</li> <li>• L’intento è quello di fornire al committente software funzionante in modo veloce. Il committente può così proporre modifiche o nuovi requisiti da integrare in iterazioni successive</li> <li>• Uno degli obiettivi delle metodologie agili è quello di mantenere al minimo la gestione “burocratica” evitando compiti e lavori di dubbio valore a lungo termine</li> </ul>	

## Manifesto per lo sviluppo agile del SW

Pur considerando il valore delle voci a destra, riteniamo più importanti le **voci a sinistra**.

<b>Individui e le interazioni</b>	Strumenti e processi
<b>Software funzionante</b>	Documentazione esaustiva
<b>Collaborazione col cliente</b>	Negoziazione dei contratti
<b>Rispondere al cambiamento</b>	Seguire un piano



(<http://agilemanifesto.org>)

## Manifesto per lo sviluppo agile del SW

### • 1. Principio

**La nostra massima priorità è soddisfare il cliente rilasciando software di valore, fin da subito e in maniera continua.**

- Ci sono varie pratiche che hanno impatto sulla qualità (MIT Sloan Management Review):
  - “Less functional the initial delivery, the higher the quality in the final delivery”
  - “The more frequent the deliveries, the higher the final quality”

## Manifesto per lo sviluppo agile del SW

### • 2. Principio

**Accogliamo i cambiamenti nei requisiti, anche a stadi avanzati dello sviluppo. I processi agili sfruttano il cambiamento a favore del vantaggio competitivo del cliente.**

- Si tratta di un'attitudine: accogliere cambiamenti significa riconoscere di aver migliorato la propria conoscenza del dominio applicativo e delle reali necessità del software

## Manifesto per lo sviluppo agile del SW

### • 3. Principio

**Rilasciamo software funzionante in modo frequente, con iterazioni variabili da un paio di settimane a un paio di mesi, preferendo i periodi brevi.**

- Rilasciare software **funzionante** subito, già dopo la prima iterazione, e di frequente.
- Ogni iterazione deve portare a nuova funzionalità che soddisfi le necessità del cliente.
- Il test deve essere il più possibile automatizzato, per non rallentare i rilasci

## Manifesto per lo sviluppo agile del SW

### • 4. Principio

**Committenti e sviluppatori devono lavorare insieme per tutta la durata del progetto.**

- Un progetto software dev'essere continuamente guidato, perché la sua direzione può variare leggermente nel tempo.
- La guida non può prescindere da una stretta collaborazione tra sviluppatori, cliente, stakeholders

## Manifesto per lo sviluppo agile del SW

### • 5. Principio

**Vogliamo basare i nostri progetti su individui motivati.  
Diamo loro l'ambiente e il supporto di cui hanno bisogno  
e confidiamo nella loro capacità di portare a termine con successo il lavoro**

- Il **team di sviluppo** è il maggiore fattore di successo di un progetto
- Tutto il resto dev'essere in funzione del team e dev'essere adattato se diventa un ostacolo

## Manifesto per lo sviluppo agile del SW

### • 6. Principio

**La conversazione diretta è il modo più efficiente e più efficace per comunicare con il team e all'interno del team**

- In una gestione agile sono previsti momenti in cui gli sviluppatori conversano in modo informale sullo stato del progetto.
- La conversazione è lo strumento più efficace. Se ne possono introdurre altri (documentazione), ma la conversazione rimane l'elemento principale.

## Manifesto per lo sviluppo agile del SW

### • 7. Principio

**Il software funzionante è il principale metro di misura dello sviluppo**

- La misura del progresso avviene in termini di funzionalità utile al cliente, non in termini di fasi, di iterazioni raggiunte, di documentazione prodotta o di infrastruttura.

## Manifesto per lo sviluppo agile del SW

### • 8. Principio

**I processi agili promuovono uno sviluppo sostenibile.**

**Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere indefinitamente un ritmo costante**

- Un progetto agile non va visto come uno scatto da 100 m, ma come una maratona. Si parte in modo sostenuto, ma costante e si cerca di mantenere la stessa velocità per l'intero percorso.
- Il team deve regolarsi in modo tale da evitare scorciatoie, burnout e garantire la migliore qualità per l'intero progetto (e per il team).

## Manifesto per lo sviluppo agile del SW

### • 9. Principio

**La continua attenzione all'eccellenza tecnica e al buon design esaltano l'agilità**

- Il modo migliore per essere veloci e garantire qualità è quello di mantenere il software più pulito e robusto possibile
- Se si danneggia o peggiora in qualche modo il codice, si deve provvedere subito a ripulire o riparare al danno

## Manifesto per lo sviluppo agile del SW

### • 10. Principio

#### La semplicità è essenziale

- Il team in un progetto agile non cerca di realizzare ciò che non è richiesto, ma segue la via più semplice, coerente con l'obiettivo di funzionalità.
- Realizzare la cosa più semplice oggi, con la massima qualità, permetterà di cambiarla velocemente domani, se sarà necessario.

## Manifesto per lo sviluppo agile del SW

### • 11. Principio

#### Le architetture, i requisiti e la progettazione migliori emergono da team che si auto-organizzano

- Le responsabilità vanno comunicate al team, non ai singoli individui.
- Il team sa auto-organizzarsi e determinare la via migliore per ottenere i risultati.
- Ogni membro di team in un progetto agile è in grado di lavorare ad ogni aspetto e fase del progetto.

## Manifesto per lo sviluppo agile del SW

### • 12. Principio

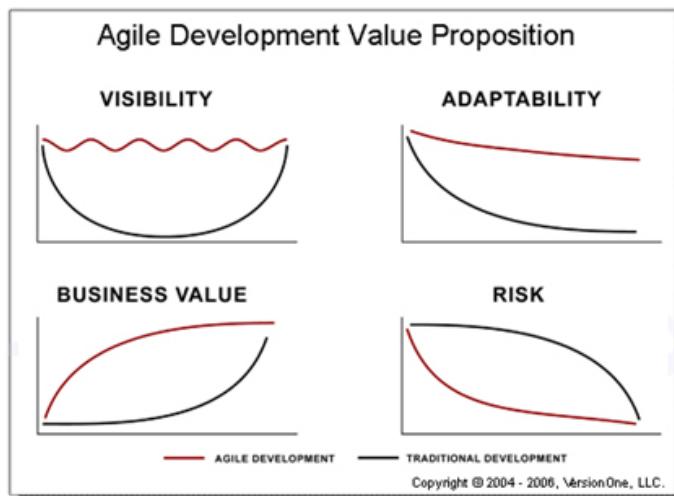
**A intervalli regolari il team riflette su come diventare più efficace, dopodiché regola e adatta il proprio comportamento di conseguenza**

- Il team rivede regolarmente organizzazione, regole interne, processi, infrastruttura, tecnologia, ecc.
- Solo adattandosi all'ambiente che cambia si riesce a rimanere agili.

## Punti di forza

- **Flessibilità nei confronti dei cambiamenti di requisiti**
- **Deliverables regolari e pianificati**
- **Coinvolgimento importante del committente**
- **Fixed budget, fixed timeline**
- **Rischio ridotto**

## Punti di forza (2)



## Valori di XP

- **Semplicità**
  - Qual è la soluzione più semplice funzionante?
  - Una soluzione semplice oggi sarà più semplice da modificare domani (piuttosto che cercare di prevedere ogni possibile opzione futura)

- **Comunicazione**

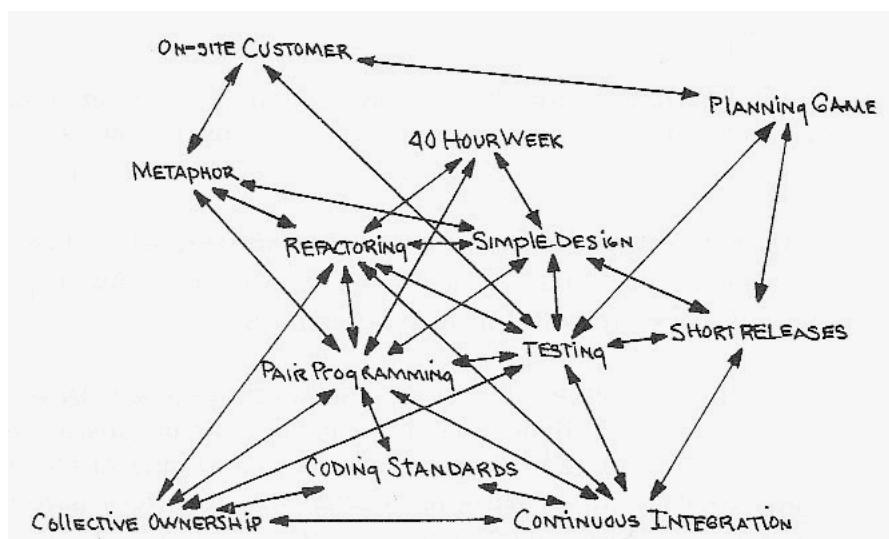
- Molti problemi che si riscontrano in un progetto sono da ricondurre a mancanza di comunicazione
- Le “buone abitudini” definite in XP obbligano alla comunicazione

- **Feedback**

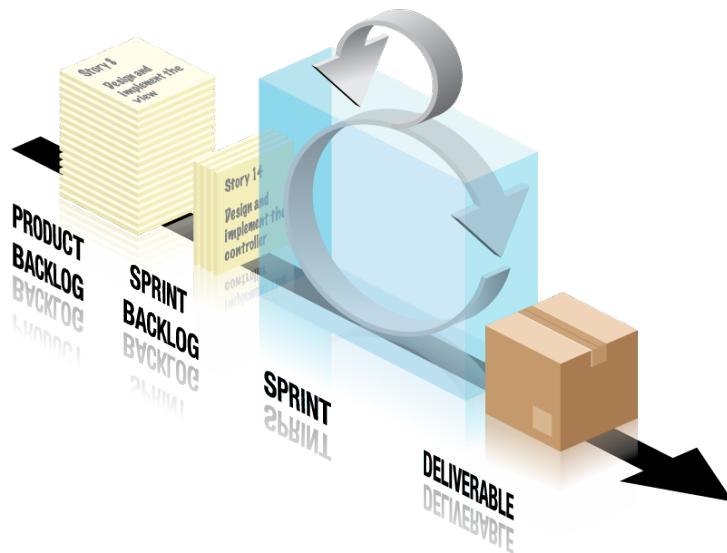
- Feedback sullo stato del sistema (test di unità, integrazione continua)
- Feedback sull'utilizzabilità del sistema (velocemente in produzione)
- Feedback sull'andamento del progetto (pianificazione e controllo)

- **Coraggio**

- Coraggio di effettuare cambiamenti in un codice funzionante
- Coraggio di condividere la paternità del codice
- Coraggio di gettare codice



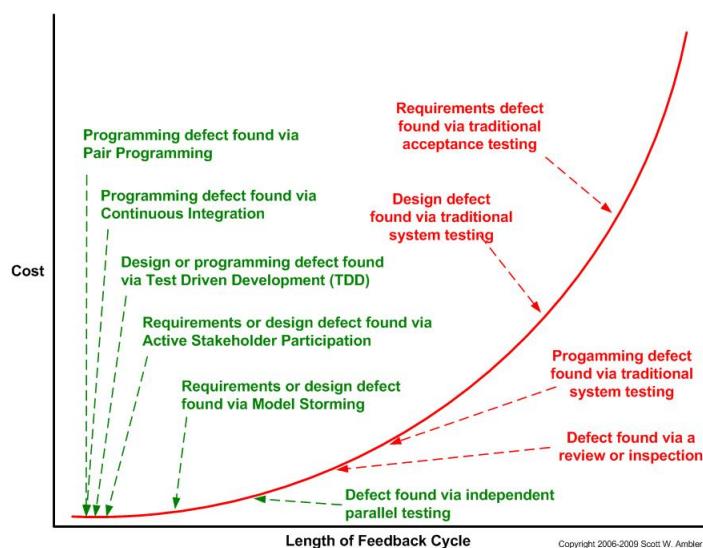
## Modello agile con Scrum



Sandro Pedrazzini

Progettazione agile nello sviluppo SW 45

## Confronti



Sandro Pedrazzini

Progettazione agile nello sviluppo SW 46

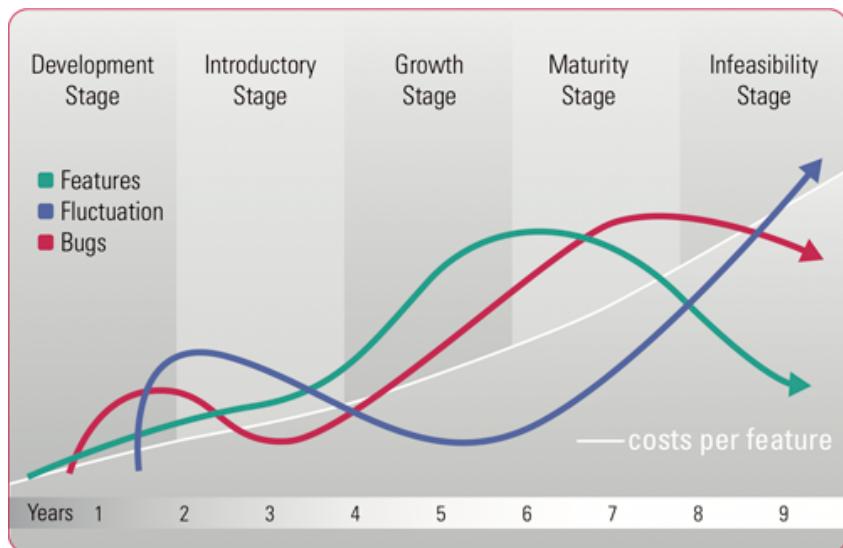
## Technical Debt

- La fretta nello sviluppo di nuove funzionalità porta a dimenticarsi di inserire nelle stime dei costi quello della manutenzione “ordinaria”
- È facile tralasciare questo aspetto
- Se non si è abituati a ripulire continuamente il codice si incorre presto nel problema chiamato “debito tecnico”

## Technical Debt (2)

- Motivo per chiamarlo “debito”: a un certo punto andrà restituito
- Più si aspetta, maggiore sarà l’interesse accumulato
- Non solo interesse: più è alto il debito, più alta è la probabilità di aggiungerne altro

### Technical Debt (3)



### Technical Debt (4)

- Il debito è il risultato dell'entropia del software che non viene combattuta in modo attivo**
- Quando si applicano le metodologie agili è importante essere vigili e gestire in modo attivo l'evoluzione dell'architettura**

## Technical Debt (5)

*On a long term project / large product, Scrum (done poorly) creates technical debt.*

*Too much technical debt left unpaid results in a Legacy System.*

Clinton Keith, The Art and Business of Making Games

## Legacy

- **Racconto di un cliente:**

- Il codice con il tempo era diventato incomprensibile
- Aggiungere al sistema nuove features era ogni volta un rischio e in ogni caso era un lavoro lungo e difficile
- C'erano bug difficili da identificare e rimuovere
- Con il vecchio sistema ci vuolevano settimane per un nuovo deployment

**=> Nuovo sistema**

**Legacy (2)****• Domande:**

- Chi ha lavorato al vecchio sistema? Si trattava di architetti e sviluppatori esperti?
- Hanno sviluppato intenzionalmente un sistema legacy?
- Cosa avete pianificato di fare in modo diverso con il nuovo sistema?
- Come assicuriamo che non diventi legacy anche il nostro?

**Legacy (3)**

- Se ci si concentra solo sulle nuove funzionalità, senza combattere in modo attivo il technical debt, si ottiene un sistema legacy
- La qualità ha un costo: meglio pagarlo subito, durante lo sviluppo delle singole storie, piuttosto che accumularlo e pagarne gli interessi

## Refactoring

- L'unico modo per evitare che l'entropia del software porti al technical debt è far uso attivo di refactoring
  - Eliminare codice inutilizzato
  - Ridurre complessità (=> ridurre linee di codice)
  - Eliminare duplicazioni
  - Facilitare collective code ownership
  - Adattare l'architettura appena si intuisce che c'è qualcosa di meglio
  - Modificare l'architettura appena si intuisce che per un certo cambiamento da implementare, una nuova struttura sarebbe più adatta

## Refactoring (2)

- Poiché refactoring non aggiunge nuove funzionalità al sistema il management non riesce a vederne il valore
- Come spiegarlo?

**Il suo valore è quello di impedire l'accumulo di debito, permettendo una velocità di sviluppo costante e "all'infinito"**

- **La gestione della visibilità che ha il cliente sul progetto non è sempre facile**
  - Lista di modifiche che cresce con velocità maggiore di quanto scenda quella delle funzionalità
  - Tentazione del cliente di intervenire sulle stime
  - Distinzione tra piccoli e grossi cambiamenti

- **Un processo adattivo richiede una relazione diversa con il cliente**
- **Un contratto a prezzo fisso lo si stipula quando i requisiti sono fissi e il margine di adattabilità è minimo**
- **Lavorando con processi adattivi, anche il prezzo va discusso in modo diverso**
- **Questo non significa che non si possa stabilire un budget per il progetto: significa che non si può fissare tempo, prezzo e funzionalità contemporaneamente**

**Cliente (2)**

- L'approccio “agile” usuale consiste nel fissare prezzo e tempo, permettendo alle funzionalità (requisiti) di variare in modo controllato

## – Esempio contratto 1:

- » scope management, basato sulle stime iniziali con sostituzione di storie
- » acceptance tests, definiti dall'utente

## – Esempio contratto 2:

- » buffer prestabilito per aggiunta di nuove funzionalità o sostituzione con storie di maggior costo

## – Esempio contratto 3:

- » time boxed

**Cliente (3)**

...

**3.4.2 Scope Changes**

If during project execution new stories need to be implemented the procedure is as following:

- 1) The new stories are estimated and categorized according to the stories of the agreed scope.
- 2) The customer goes through the remaining scope (work not started) and tries to remove stories in the same value like the new scope.
  - a. If this is possible, these changes are noted in the Scope Management list. The overall amount of work has not changed and the new stories are treated like they have been there from the beginning.
  - b. If this is not possible, the new stories are listed in the Scope Management list as extensions and are treated as new stories.

...

- Ad ogni iterazione il cliente ha la possibilità di vedere una nuova funzionalità integrata nel programma finale e può nel contempo intervenire sulla direzione da prendere per l' iterazione seguente
- Il software, utilizzabile, anche se non completo, può andare in produzione molto presto
- Nel processo predittivo l' unico criterio di qualità è il confronto con il piano del progetto
- Nel processo adattivo, il piano va rivisto ad ogni iterazione. I rischi sono mantenuti ai minimi termini

**Anche il controllo se un progetto ha avuto successo o meno è diverso:**

- In un processo predittivo si misura se i risultati sono in base ai piani. Se poi si è in tempo e nel budget, il progetto viene considerato un successo
- In un processo adattivo, il criterio principale è il valore di business. Il progetto è un successo se ha mantenuto i parametri di tempo e prezzo, ma ha prodotto un programma diverso e migliore di quello previsto

## Ruolo dello sviluppatore (1)

- L'esecuzione di un progetto con metodi agili richiede un team di sviluppo con persone ben formate e ben integrate tra di loro.
- Pare esserci una sinergia interessante (Fowler): non solo l'adattabilità ha bisogno di un buon team, ma buoni sviluppatori tendono a preferire metodologie agili.
- Lo sviluppatore è attivo sia nella fase di analisi, che in quelle del design, della codifica e del test.

## Ruolo dello sviluppatore (2)

- Allo sviluppatore è richiesto di prendere decisioni tecniche e di stimare il tempo necessario per un certo lavoro.
- La suddivisione dei compiti all'interno del progetto è più orizzontale e meno gerarchica
- Durante lo svolgimento del progetto gli sviluppatori devono rimanere in stretto contatto con il cliente (business expertise)

## Adottare metodologie agili

**"Agile" is a marketing term created to describe a style of working. This style focuses on collaborative work, concrete results, delivering value, and minimizing waste.**

**"Being agile" means working in the agile style. To do so, start with an existing agile method, use it to understand the principles and practices of agile development, then adapt it to your specific situation.**

James Shore, The Art of Agile Development

## Riferimenti

Beck K.: "Test-Driven Development", Addison Wesley, 2003

Dräther, Koschek, Sahling: "Scrum – kurz & gut", O'Reilly 2013

Fowler M. : "The New Methodology", <http://www.martinfowler.com/>, 2003

Martin R.C.: "Clean Code: A Handbook of Agile Software Craftsmanship", Prentice Hall, 2008

Pedrazzini S.: "Tecniche di progettazione agile con Java: Design pattern, refactoring e test", Tecniche nuove, 2006

Shore J.: "The Art of Agile Development", O'Reilly 2007

Sommerville J.: "Software Engineering", 9<sup>th</sup> Edition, Addison-Wesley, 2011

# Software Engineering: Test

## 1. Incremental development

- 1) Verify your *JUnit* installation and realize the exercise incrementally, code a little test a little, using *JUnit* for whatever test and after each single function's implementation.
- 2) Define, in a class named "Worker", a simple static method named "conversion", which receives an integer value as parameter and returns half of that value if the value is even, or three times plus one of that value if the value is odd.

### **Test its behavior.**

Now, write a second method, named "sequence", with an integer parameter named "startingValue", able to iteratively apply the first method ("conversion") and determine how long is the sequence of repeated calls of "conversion" until you get the value 1, starting from whatever integer value ("startingValue") > 2 (for "repeated calls" we mean that the returned value of the first "conversion" call is used as input for the next "conversion" call, and so on).

**Test its behavior and test that if the starting value is <=2 the "sequence" method throws an exception.**

Example of sequence(10):

Starting value = 10

Obtained values through "conversion": 10 => 5 16 8 4 2 1

Length: 6

- 3) Refactor your class, so that "sequence" has no parameter anymore, but the "startingValue" is defined as field of "Worker" and is initialized through the constructor:

```
Worker worker = new Worker(10);
worker.sequence(); // == 6
```

You need a new object for each new sequence.

**Adapt your tests, to verify that this new version still works correctly.**

- 4) Write a new class "SequenceCache", able to return the sequence for each value, using an internal Map to cache the pairs <value, Worker object>.

The value is specified as integer parameter of the method “length”. If for a certain value the corresponding Worker already exists in the Map, the “length” method internally calls the Worker method “sequence” on it and returns the sequence value.

If, however, there is no Worker, the class creates a new corresponding worker first, adds it to the Map, and then calls the “sequence” method on it.

```
SequenceCache sc = new SequenceCache();
sc.length(10); // == 6
```

**Test the behavior of the new class “SequenceCache”, also considering exceptional cases.**

## 2. Fraction

- 1) Define the class “Fraction”, representing a fraction as pair of longs. Implement the 4 base operations, starting from the addition.  
Implement the “simplify” operation, using the greatest common divisor (gcd) function.

Remember: provide the equals method for Fraction, able to compare for equality two distinct, but equal, objects.

Note:

the greatest common divisor (gcd), also known as the greatest common factor (gcf), highest common factor (hcf), or greatest common measure (gcm), of two or more integers (at least one of which is not zero), is the largest positive integer that divides the numbers without a remainder. For example, the GCD of 8 and 12 is 4.

$$\begin{aligned} \text{gcd}(u,0) &= u \\ \text{gcd}(u,v) &= \text{gcd } (v,u) \\ \text{gcd } (u,v) &= \text{gcd } (v,u\%v) \end{aligned}$$

Least common multiple:  
 $\text{lcm}(u,v) = u * v / \text{gcd } (u,v)$ .

- 2) Implements in *JUnit* all test incrementally, demonstrating the correct behavior of the class, also considering exceptional cases (like division by zero).

## Test

### Test design and unit testing

### The problem

- Each developer knows that he should write testing code, but he rarely does it.
- Why?
  1. Hurry, delivery deadlines
  2. Too much self confidence (??)
  3. Wrong education
  4. Lack of professionalism

## Causes (1)

### Hurry, delivery deadlines

You enter into a vicious circle. The higher the haste, the less you invest time to write the test. Fewer tests you write and less stable is the code. If the code is less stable you will lose time and therefore haste increases again, ...

### Too much self confidence

Wrong attitude for SW developers. You must feel the "pleasure" of finding errors in your code

-> "programmers love writing tests"

## Causes (2)

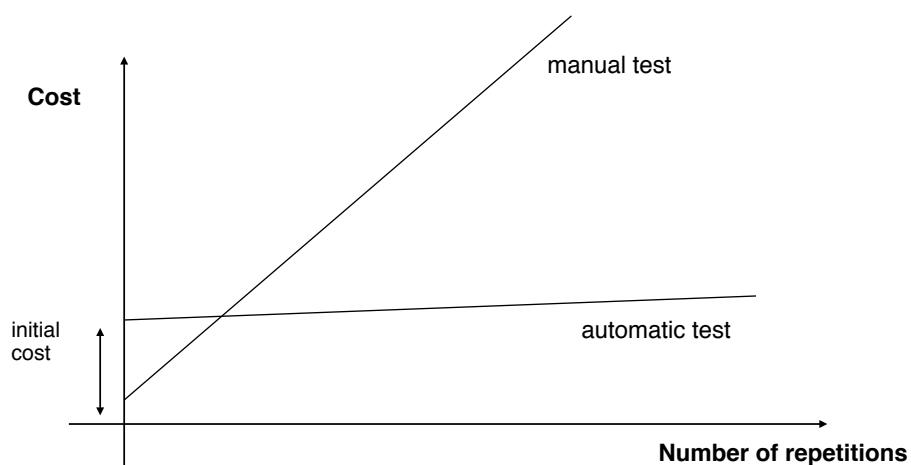
### Wrong education

Developing SW is a complex task.  
It is not like solving an equation with one unique solution.  
You get many solutions and you are finding new details again and again.  
Only experienced SW developers should teach programming.

### Lack of professionalism

The errors' search is part of the development activity.  
Programming goes strictly together with testing.  
"I write the program, then someone else should test it..."

- Test shortens the development cycles, where a cycle is defined as a period in which the code changes maintaining at the end a consistent state (maximum one day, usually minutes or hours).  
-> “code a little, test a little”
- It increases the self-esteem and confidence in your software
- It increases the productivity
- It predisposes you to change: Never say: “It works fine, so I do not change it ...”
- It facilitates and leads to the concepts of refactoring and "program to change"



## XP: extreme programming

**Concept of test brought to its extreme  
(also in TDD, Test Driven Development)**

**The test is the core of development:**

I write the test first,  
then the code that can run it correctly.

**Development tools for XP:**

From the test code the tool can generate “stubs”, i.e. the missing pieces of code.

Maximal automatization of refactoring’s elements.

## From agile manifesto

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
- Working software is the primary measure of progress

## How we should write tests

- From detail to more general tests (bottom-up)
- Accumulate old tests and recall them regularly
- The control of correctness must be automatic (avoid, if possible, to manually analyze long output)
- Easy to write, without "damage" too much the code of the program to be developed
- The realization of the test can anticipate the code to be developed (XP)

## How to facilitate test in Java (1)

```
public class Arc {  
    ...  
    public void debug(){  
        System.err.println("Content: " + getInfo());  
        if (nextStates != null)  
            nextStates.debug();  
    }  
}
```

**NO !**

- Use a debugger for such purposes
- Do not mix test and productive code

## How to facilitate test in Java (2)

```
public class Traverse {
    ...
    public static void main(String[] args){
        Traverse traverse = new Traverse();
        ...
        traverse.execute(structure);
        ...
        if (traverse.isOk())
            System.err.println("Execution ok");
        else
            System.err.println("Error detected");
    }
}
```

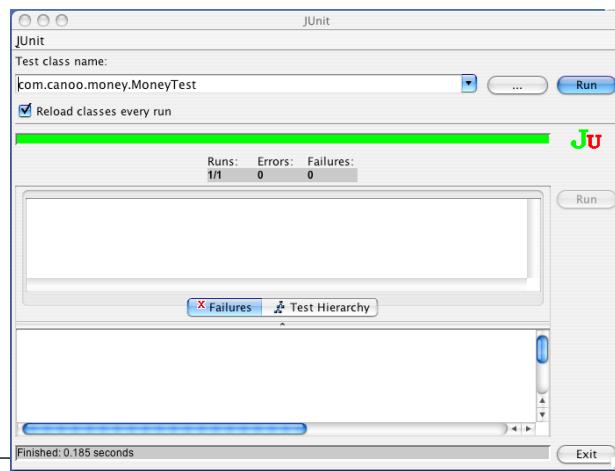
**NO !**

- **Distributed code**
- **Do not mix test and productive code**

## How to facilitate test in Java (3)

Use a tool that will facilitate the implementation of test and the result check.

**Example:** JUnit



## First example with JUnit (1)

### Program “Calculator”

```
public class Calculator{
    public int multByTwo(int value) {
        return value * 2;
    }
}

public class Main {
    ...
    public static void main( String[] args ) {
        Calculator calculator = new Calculator();
        System.out.println(calculator.multByTwo(5));
    }
}
```

## Test Example

### Test of “Calculator”

```
public class CalulatorTest {
    ...

    @Test
    public void testMultByTwo () {
        Calculator calculator = new Calculator();

        assertEquals(0, calculator.multByTwo(0));
        assertEquals(10, calculator.multByTwo(5));
        assertEquals(-10, calculator.multByTwo(-5));
    }
}
```

assert methods statically imported

```
import static org.junit.Assert.*;
```

## Running it

### How to run a test class/suite

#### 1. IDE integration (the most used)

#### 2. Method: *main()*

```
public static void main(String args[]) {
    junit.ui.TestRunner.run(HelloWorldTest.class);
}
```

Call: java HelloWorldTest

#### 3. Metodo: *suite()*

```
public static Test suite(){
    return new TestSuite(HelloWorldTest.class);
}
```

Call: java junit.swingui.TestRunner HelloWorldTest

## Assert

static void	<b>assertEquals</b> (java.lang.String message, long expected, long actual) Asserts that two longs are equal.
static void	<b>assertEquals</b> (java.lang.String message, java.lang.Object[] expecteds, java.lang.Object[] actuals) <i>Deprecated. use assertEqualsArrayEqual</i>
static void	<b>assertEquals</b> (java.lang.String message, java.lang.Object expected, java.lang.Object actual) Asserts that two objects are equal.
static void	<b>assertFalse</b> (boolean condition) Asserts that a condition is false.
static void	<b>assertFalse</b> (java.lang.String message, boolean condition) Asserts that a condition is false.
static void	<b>assertNotNull</b> (java.lang.Object object) Asserts that an object isn't null.
static void	<b>assertNotNull</b> (java.lang.String message, java.lang.Object object) Asserts that an object isn't null.
static void	<b>assertNotSame</b> (java.lang.Object unexpected, java.lang.Object actual) Asserts that two objects do not refer to the same object.
static void	<b>assertNotSame</b> (java.lang.String message, java.lang.Object unexpected, java.lang.Object actual) Asserts that two objects do not refer to the same object.
static void	<b>assertNull</b> (java.lang.Object object) Asserts that an object is null.
static void	<b>assertNull</b> (java.lang.String message, java.lang.Object object) Asserts that an object is null.
static void	<b>assertSame</b> (java.lang.Object expected, java.lang.Object actual) Asserts that two objects refer to the same object.
static void	<b>assertSame</b> (java.lang.String message, java.lang.Object expected, java.lang.Object actual) Asserts that two objects refer to the same object.
static <T> void	<b>assertThat</b> (java.lang.String reason, T actual, org.hamcrest.Matcher<T> matcher) Asserts that actual satisfies the condition specified by matcher.
static <T> void	<b>assertThat</b> (T actual, org.hamcrest.Matcher<T> matcher) Asserts that actual satisfies the condition specified by matcher.
static void	<b>assertTrue</b> (boolean condition) Asserts that a condition is true.
static void	<b>assertTrue</b> (java.lang.String message, boolean condition) Asserts that a condition is true.

## More complex example

### Development

Interrelationship between development and testing.  
I write a few lines of code, then I write and execute test

-> “code a little, test a little”

### Problem

Money sums and representations with more than one currency. .

The additions between elements of the same currency are simple numerical sums or subtraction, without the need to consider the currency used.

Things get more interesting if there are multiple currencies involved.  
You cannot just add or subtract them because the conversion rate is not fixed.  
I might want to have the value in CHF of a combination of different currencies at the current value, at that of yesterday, a month ago, and so on.

## class Money

We begin by defining the *Money* class, which manages the value of a single currency. The value is an *int*, while the currency is represented by a string of 3 letters (ISO abbreviation).

```
public class Money {
    private int amount;
    private String currency;

    public Money(int amount, String currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public int getAmount() {
        return amount;
    }

    public String getCurrency() {
        return currency;
    }
}
```

## Equality

We want first to specify the equality between two distinct objects with the same values.

The following method is used to define the equality between two *Money* objects (overriding of *equals()*):

```
public boolean equals(Object other) {  
    return ((other instanceof Money) &&  
            ((Money)other).getCurrency().equals(currency) &&  
            ((Money)other).getAmount() == amount);  
}
```

## First test

It is already time to write the first test...

```
@Test  
public void testEquals() {  
    Money m12CHF = new Money(12, "CHF"); // (*)  
    Money m14CHF = new Money(14, "CHF");  
  
    assertEquals(m12CHF, new Money(12, "CHF"));  
    assertFalse(m12CHF.equals(m14CHF));  
}
```

### Necessary:

Initialization code (\*), code with the classes to verify and code executing the check ("assert...").

**Add**

The sum of two values of the same currency corresponds to an object of type *Money* that has as its value the whole sum of the two operands:

```
public Money add(Money money) {  
    return new Money(amount + money.getAmount(), currency);  
}
```

**Test for *add()***

Here our test for the sum operation.

```
public class MoneyTest {  
    ...  
  
    @Test  
    public void testSimpleAdd() {  
        Money m12CHF= new Money(12,"CHF");  
        Money m14CHF= new Money(14,"CHF");  
        Money expected= new Money(26,"CHF");  
  
        assertEquals(expected, m12CHF.add(m14CHF));  
    }  
}
```

## Initializations (1)

Looking at the first two test functions, we can observe that there are elements in common: the elements of initialization

```
Money m12CHF= new Money(12, "CHF");  
Money m14CHF= new Money(14, "CHF");
```

JUnit provides a "fixture" mechanism for setup (@Before or @BeforeEach (from v5))

The *@BeforeEach* method is called before each test, ensuring no side effects.

The symmetric mechanism is the tear down (@After or @AfterEach)

## Initializations (2)

```
public class MoneyTest {  
    private Money f12CHF;  
    private Money f14CHF;  
  
    @BeforeEach  
    protected void setUp() {  
        f12CHF= new Money(12, "CHF");  
        f14CHF= new Money(14, "CHF");  
    }  
    ...  
}
```

Initialization method

## Initializations (3)

We can adapt the two test methods as follows:

```
@Test
public void testEquals() {
    assertEquals(f12CHF, f12CHF);
    assertEquals(f12CHF, new Money(12, "CHF"));
    assertFalse(f12CHF.equals(f14CHF));
}

@Test
public void testSimpleAdd() {
    Money expected= new Money(26, "CHF");
    assertEquals(expected, f12CHF.add(f14CHF));
}
```

## Test Suite

If you need to combine the execution of different classes, you can create a Suite with the annotations `@RunWith` and `@Suite`

The IDE integrations already allow combination of classes

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    CalculatorTest.class
    SquareTest.class
})
public class AllTests {}
```

**Execution** **SUPSI**

### Run (1)

JUnit can be called in several ways. We are interested in the way that shows the result within a GUI.

```
java junit.swingui.TestRunner
```

The screenshot shows the JUnit Swing GUI window. At the top, there's a toolbar with icons for file operations like Open, Save, and Print. Below the toolbar, a menu bar has "File" and "Help" options. The main area is titled "JUnit" and contains a "Test class name:" dropdown set to "com.canoo.money.MoneyTest", a checked checkbox for "Reload classes every run", and a "Run" button. Below this is a progress bar with the letters "JU" in red. Underneath the progress bar, status information shows "Runs: 1/1 Errors: 0 Failures: 0". The bottom of the window displays a message "Finished: 0.185 seconds" and buttons for "Run" and "Exit".

Sandro Pedrazzini Test 27

**Execution** **SUPSI**

### Integration in IDEs (1)

- Eclipse

The screenshot shows the Eclipse IDE interface with the "Package Explorer" view on the left and the "JUnit" view on the right. The "JUnit" view displays the results of a run: "Finished after 0.053 seconds", "Runs: 5/5 (1 ignored)", "Errors: 0", and "Failures: 0". A green progress bar at the bottom of the "JUnit" view indicates success. Below the progress bar, a list shows "CalculatorTest [Runner: JUnit 4]".

Sandro Pedrazzini Test 28

**Execution**

**SUPSI**

## Integration in IDEs (2)

- IDEA

```

All 10 tests passed - 19s 703ms
/Library/Java/JavaVirtualMachines/jdk1.7.0_13.jdk/Contents/Home/bin/java ...
/Users/sandro/TransducerDeveloping/tests/license.cfg
...Checking License File /Users/sandro/TransducerDeveloping/tests/license.cfg
Warning: error creating a server connection.
...
New internal cache size: 20000
/Users/sandro/TransducerDeveloping/tests/license.cfg
...Checking License File /Users/sandro/TransducerDeveloping/tests/license.cfg
Warning: error creating a server connection.
Warning: error creating a server connection.
Query: aufsinken
Form: aufsinken
Inflection information:
(Cat V)(Aux sein)(Mod Inf)(Temp Pres)(ID 0-1)
(Cat V)(Aux sein)(Mod Ind)(Temp Pres)(Pers 1st)(Num Pl)(TD 0-1)
...

```

Tests Passed: 10 passed (moments ago)

Sandro Pedrazzini

Test 29

**Example (cont.)**

**SUPSI**

## MoneyBag (1)

After verifying the execution of Money, we can work on cases with more currencies.

**Problem**

There is no single reference exchange.

**Solution**

Delaying the conversion, recording all the different elements.

**Example**

The sum of 10 CHF and 10 USD must result in a bag containing the two separate values.

If we add additional 20 CHF to the bag the resulting bag must now contain 30 CHF and 10 USD.

Sandro Pedrazzini

Test 30

## MoneyBag (2)

The *MoneyBag* class manages the different currencies.

```
public class MoneyBag {
    private List<Money> monies= new ArrayList<>();

    public MoneyBag(Money m1, Money m2) {
        appendMoney(m1);
        appendMoney(m2);
    }

    public MoneyBag(Money bag[]) {
        for (Money money : bag) {
            appendMoney(money);
        }
    }
    ...

    private void appendMoney(Money money){
        monies.add(money);
    }
    ...
}
```

## MoneyBag Test

Also for *MoneyBag* we must implement *equals()*, to define the equality criterion.

Then we can verify the class by adding a new test method:

```
@Test
public void testBagEquals() {
    Money m12CHF = new Money(12, "CHF");
    Money m7EUR = new Money(7, "EUR");
    Money m21EUR = new Money(21, "EUR");

    MoneyBag mb1 = new MoneyBag(m12CHF, m7EUR);
    MoneyBag mb1reverted = new MoneyBag(m7EUR, m12CHF);
    MoneyBag mb2 = new MoneyBag(m12CHF, m21EUR);

    assertEquals(mb1, new MoneyBag(m12CHF, m7EUR));
    assertEquals(mb1, mb1reverted);

    assertFalse(mb1.equals(mb2));
}
```

## Method *add()*

We can now generalize the Money *add()* method.

### Initial version

```
public Money add(Money money) {
    return new Money(amount + money.getAmount(), currency);
}
```

### First variant

```
public Money add(Money money) {
    if (money.getCurrency().equals(currency)){
        return new Money(amount + money.getAmount(), currency);
    }

    return new MoneyBag(this, money);
}
```

## IMoney

The previous method definition cannot be compiled, because there is no relationship between *Money* and *MoneyBag*.

We have two *Money* representations that we would like to bind under a unique interface:

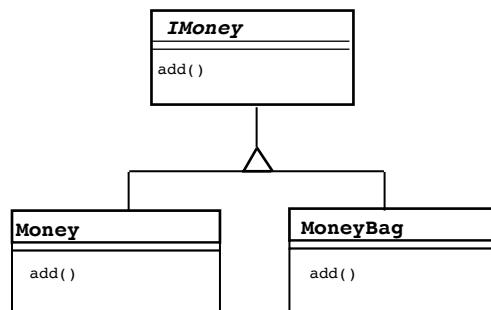
```
public interface IMoney {
    IMoney add(IMoney money);
}

public class Money implements IMoney
{...}

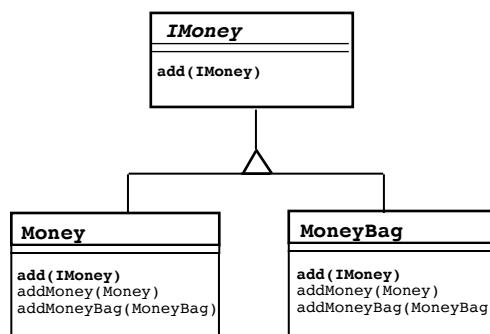
public class MoneyBag implements IMoney
{...}
```

**IMoney (2)**

With *IMoney* we want to get an interface that allows us to hide the use of *Money* and *MoneyBag*.

**IMoney (2)**

- But we cannot dispatch on the parameter, so let's implement first the different combinations of *add()*, naming them *addMoney()* and *addMoneyBag()*, for clearness.



## Implementation in Money

```

public class Money implements...
...
protected IMoney addMoney(Money money) {
    if (money.getCurrency().equals(currency)){
        return new Money(ammount + money.getAmount(), currency);
    } else {
        return new MoneyBag(this, money);
    }
}

protected IMoney addMoneyBag(MoneyBag bag) {
    return new MoneyBag(bag, this);
}

...
}

```

Sandro Pedrazzini

Test 37

## Test of Money (1)

### 1. Sum of two Money elements

```

@Test
public void testMoneyToMoney() {
    // [12 CHF] + [7 EUR] == {[12 CHF][7 EUR]}
    Money m12CHF = new Money(12, "CHF");
    Money m7EUR = new Money(7, "EUR");

    MoneyBag expected= new MoneyBag(new Money[]{m12CHF, m7EUR});

    assertEquals(expected, m12CHF.addMoney(m7EUR));
}

```

Sandro Pedrazzini

Test 38

## Test of Money (2)

### 2. Add a MoneyBag to a Money

```
@Test
public void testAddMoneyBagToMoney() {
    ...
    MoneyBag expected =
        new MoneyBag(new Money[ ]{m12CHF, m7EUR, m20USD});
    MoneyBag second = new MoneyBag(m12CHF, m7EUR);

    assertEquals(expected, m20USD.addMoneyBag(second));
}
```

## Implementation in MoneyBag

**Implementation in MoneyBag**  
 (further MoneyBag constructors should be provided):

```
public class MoneyBag extends ...
    ...
    protected IMoney addMoney(Money money) {
        return new MoneyBag(this, money);
    }

    protected IMoney addMoneyBag(MoneyBag bag) {
        return new MoneyBag(bag, this);
    }

    ...
}
```

## Test of MoneyBag (1)

### 3. Add a Money to a MoneyBag

```
@Test
public void testAddMoneyToMoneyBag() {
    Money m12CHF = new Money(12, "CHF");
    Money m7EUR = new Money(7, "EUR");
    Money m20USD = new Money(20, "USD");

    MoneyBag expected =
        new MoneyBag(new Money[]{m12CHF, m7EUR, m20USD});
    MoneyBag first = new MoneyBag(m12CHF, m7EUR);

    assertEquals(expected, first.addMoney(m20USD));
}
```

## Test of MoneyBag (2)

### 4. Add a MoneyBag to a MoneyBag

```
@Test
public void testAddMoneyBagToMoneyBag() {
    Money m12CHF = new Money(12, "CHF");
    Money m5CHF = new Money(5, "CHF");
    Money m20EUR = new Money(20, "EUR");
    Money m7USD = new Money(7, "USD");
    Money m17CHF = new Money(17, "CHF");

    MoneyBag first = new MoneyBag(new Money[]{m7USD, m12CHF});
    MoneyBag second = new MoneyBag(new Money[]{m20EUR, m5CHF});
    MoneyBag expected =
        new MoneyBag(new Money[]{m17CHF, m7USD, m20EUR});

    assertEquals(expected, first.addMoneyBag(second));
}
```

## Double “dispatching” (1)

### Problem:

We want to implement `add()`:

```
public IMoney add(IMoney money);
```

But we should have the double dispatching in order to be able to call:

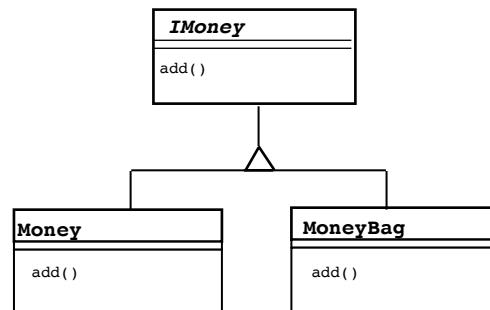
```
IMoney a = ...
IMoney b = ...

a.add(b);
```

Without having to know the specific class of objects *a* and *b*

## Simple dispatching with IMoney

```
IMoney m = ...
...
m.add(new Money(10, "CHF"));
...
```



## Double “dispatching” (3)

### Java

The double dispatching would mean that also a dispatching on the parameter should be performed: this is not possible in Java.

Very few languages go beyond the simple dispatching on the object that performs the call.

In Java the simple dispatching on the invoker object is automatic, executed by the system.

The dispatching on the parameters is not automatic.

## Double “dispatching” (4)

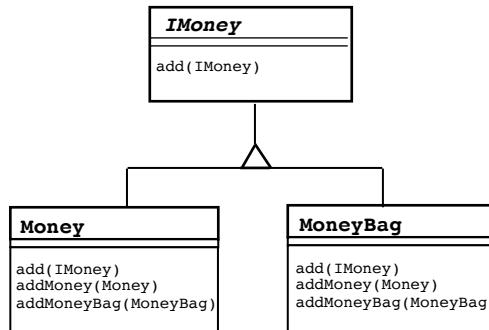
**Two ways to solve our problem:**

1. **Manual dispatching**  
Simpler in this case, but less OO.
2. **Pattern Visitor: simulation of double dispatching**  
More elegant

## Dispatching on the parameter

In any case:

we need to be able to distinguish if the parameter is a *Money* or a *MoneyBag*, because we want to pass it as an *IMoney*



## 1. „Manual“ dispatching

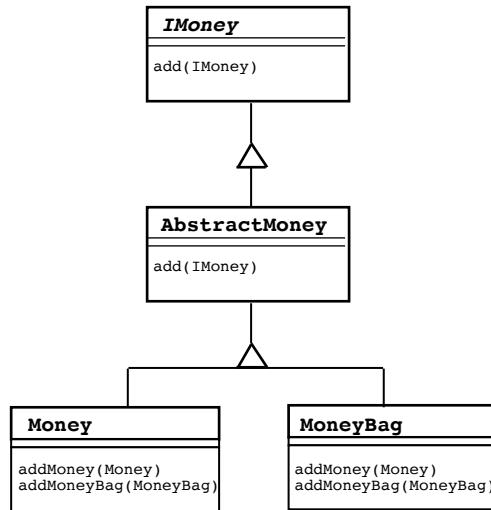
```

public IMoney add(IMoney money){
    if (money instanceof Money)
        return addMoney((Money)money);
    else
        return addMoneyBag((MoneyBag)money);
}
  
```

The implementation of this method is the same for both *Money* and *MoneyBag*. For this reason it can be plugged into an abstract common class.

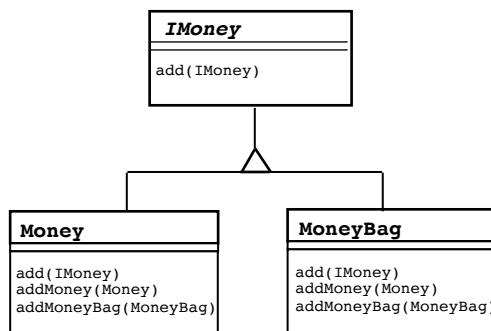
The addition of a new common class, a superclass of *Money* and *MoneyBag*, would not result in a change of the interface *IMoney*.

## „Manual“ dispatching (2)



## 2. Implementation with Visitor (1)

Since Java does not have the double dispatching, the *Visitor pattern* is a stylish alternative to easily manage combinations of *Money* and *MoneyBag* (in which both can appear as invoking object or as parameter).



## 2. Implementation with *Visitor* (2)

In *Money* e *MoneyBag* we need to implement two versions od *add()*

```
public class Money extends ...  
...  
    public IMoney add(IMoney money) {  
        return money.addMoney(this);  
    }  
}  
  
public class MoneyBag extends ...  
...  
    public IMoney add(IMoney money) {  
        return money.addMoneyBag(this);  
    }  
}
```

## 2. Implementation with *Visitor* (3)

In order to be able to perform such method calls with *IMoney* object within *add()*

```
money.addMoney(this);  
money.addMoneyBag(this);
```

*IMoney* should be extended with the declaration of two further methods, resulting in following:

```
public interface IMoney {  
    IMoney add(IMoney money);  
    IMoney addMoney(Money money);  
    IMoney addMoneyBag(MoneyBag moneyBag);  
}
```

**BUT: this is exactly what we did not want...**

## 2. Implementation with *Visitor* (4)

### Solution:

In order to avoid the definition of the two methods in the *IMoney* interface, we define an intermediate abstract class:

```
public abstract class AbstractMoney implements IMoney {
    abstract protected IMoney addMoney(Money m);
    abstract protected IMoney addMoneyBag(MoneyBag mb);
}
```

## 2. Implementation with *Visitor* (5)

### Changes in *add()* implementations:

```
public class Money extends AbstractMoney {
    ...
    public IMoney add(IMoney m) {
        AbstractMoney money = (AbstractMoney)m;
        return money.addMoney(this);
    }
}

public class MoneyBag extends AbstractMoney {
    ...
    public IMoney add(IMoney m) {
        AbstractMoney money = (AbstractMoney)m;
        return money.addMoneyBag(this);
    }
}
```

**Test (1)**

- The previously specified tests still run successfully in both versions.
- We want to enhance them, however, with the call to `add()`.
- Let's show this with the simplest case

```
@Test
public void testMoneyToMoney() {
    // [12 CHF] + [7 EUR] == {[12 CHF][7 EUR]}
    Money m12CHF = new Money(12, "CHF");
    Money m7EUR = new Money(7, "EUR");

    MoneyBag expected= new MoneyBag(new Money[]{m12CHF, m7EUR});

    assertEquals(expected, m12CHF.addMoney(m7EUR));
    assertEquals(expected, m12CHF.add(m7EUR));
}
```

**Test (2)**

- Or, even better, change the elements to `IMoney` elements

```
@Test
public void testMoneyToMoney() {
    // [12 CHF] + [7 EUR] == {[12 CHF][7 EUR]}
    IMoney m12CHF = new Money(12, "CHF");
    IMoney m7EUR = new Money(7, "EUR");

    IMoney expected= new MoneyBag(new Money[]{m12CHF, m7EUR});

    assertEquals(expected, m12CHF.add(m7EUR));
}
```

## Simplifying bags (1)

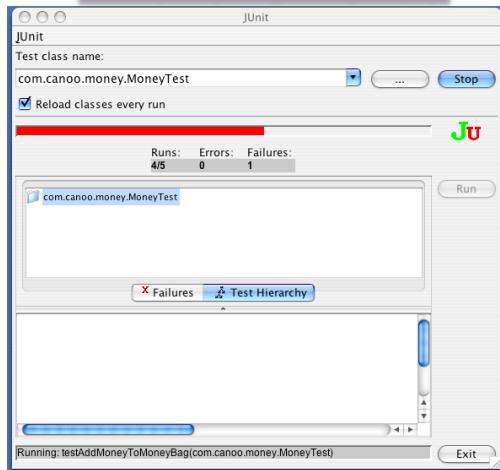
If I think of a particular new feature, I can try to write a test first, and observe the result (TDD):

```
@Test
public void testSimplify() {
    // {[12 CHF][7 USD]} + [-12 CHF] == [7 USD]

    IMoney first = new MoneyBag(new Money[]{f12CHF, f7USD});
    IMoney expected = new Money(7, "USD");

    assertEquals(expected, first.add(new Money(-12, "CHF")));
}
```

## Simplifying bags (2)



In this case the test fails. This shows me that I need to perform changes to the program, in order to make sure that the JUnit bar will turn from red to green ...

## Simplifying bags (3)

The simplification consists in the modification of the MoneyBag methods, assuming that the constructor already takes care of removing the List items with numeric value zero.

```
public IMoney addMoney(Money money) {
    return (new MoneyBag(money, this)).simplify();
}

public IMoney addMoneyBag(MoneyBag bag) {
    return (new MoneyBag(bag, this)).simplify();
}

private IMoney simplify() {
    if (monies.size() == 1) {
        return monies.get(0);
    }
    return this;
}
```

## Final remarks (1)

1. The implemented example is only part of the solution to the initial problem. It was important to show the developing approach "code a little, test a little."
  
2. We have shown the first test after realizing the first *add()* method. This is the right moment to write a test, because it is immediately after having developed a function that we know exactly what to test.  
  
**-> each new test is a reference point in the implementation of the program**

## Final remarks (2)

3. We have first created an initial test suite, on which we added new tests.  
It is important to accumulate old tests and call them regularly.

-> It represents a robustness check that helps increasing  
the developer's self confidence

4. We have created a test as soon as we had a new functionality's idea.

-> The implementation becomes the way to run the test

## Test Infected

"Test infected" persons change their attitude towards SW development.

- They are no longer able to go home if the tests do not correctly run (end of a development's cycle).
- They feel a big pleasure each time they see a green line...
- They want to infect anyone else around them (the best way is through direct contact: as soon as someone needs help in debugging, they begin to write test methods).
- They have an out-of-the-ordinary confidence in their own code and a great ability to refactor
- **They are able to provide SW quality in shorter and shorter time**

**More features**

- Exceptions' check
- Timeout
- Class fixture
- Ignore

**Initial class**

```
public class Calculator {  
    private int status = 0;  
  
    public void clear(){  
        status = 0;  
    }  
  
    public void add(int operand){  
        status += operand;  
    }  
  
    ...  
  
    public void switchOn() {           // switch on calculator  
        status = 0;  
    }  
  
    public void switchOff() { }       // switch off calculator  
  
    public int getResult() {  
        return status ;  
    }  
}
```

**Test**

```
public class CalculatorTest {  
  
    private Calculator calculator = new Calculator();  
  
    ...  
  
    @Test  
    public void add() {  
  
        calculator.add(1);  
        calculator.add(1);  
  
        assertEquals(2, calculator.getResult());  
    }  
}
```

**Fixture**

**@Before, @BeforeEach, @After, @AfterEach**

```
@Before  
public void clearCalculator() {  
    calculator.clear();  
}
```

## Exceptions' check

- Without special features

```
@Test  
public void testDevideByZero(){  
    try{  
        calculator.divide(0);  
        fail();  
    }catch(ArithmeticalException e){}  
}
```

## Exceptions' check (2)

- With annotation's parameter (v4)

```
@Test(expected = ArithmeticalException.class)  
public void divideByZero() {  
    calculator.divide(0);  
}
```

## Exceptions' check (3)

- With **assertThrows** and **lambda** (v5)

```
@Test  
public void divideByZero() {  
    assertThrows(ArithmetricException.class,  
                () -> calculator.divide(0));  
}
```

## Exceptions' check (4)

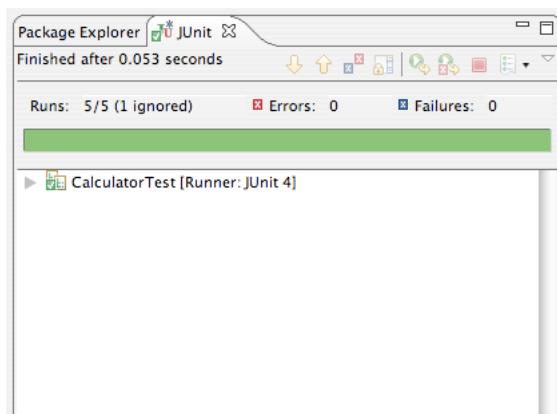
- With **message check**

```
@Test  
public void divideByZero() {  
    Exception ex = assertThrows(MyArithmetricException.class,  
                                () -> { calculator.divide(0) });  
    assertEquals("Zero not allowed", ex.getMessage());  
}
```

**Ignore**

- Test to ignore temporarily (v4: *Ignore*, v5: *Disable*)

```
@Ignore("not ready yet")
@Test
public void multiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(100, calculator.getResult());
}
```

**Integration in IDE**

## Class fixture

**@BeforeClass or @BeforeAll:** one method per class, public static, called only once for the entire class

**@AfterClass or @AfterAll:** one method per class, public static, called only once for the entire class

```
private static Calculator calculator;  
...  
@BeforeClass  
public static void switchOnCalculator() {  
    calculator = new Calculator();  
    calculator.switchOn();  
}  
  
@AfterClass  
public static void switchOffCalculator() {  
    calculator.switchOff();  
    calculator = null;  
}
```

## Timeout

- Considering that there is an error in the method `squareRoot()` which leads to an infinite loop, here's how to get out of the test (with error), after a timeout of 1 second:

```
@Test(timeout = 1000)  
public void squareRoot() {  
    calculator.squareRoot(2);  
}
```

## Timeout (2)

- With lambda (v5):

```
public void squareRoot() {  
    assertTimeout(ofSeconds(1),  
                 () -> { calculator.squareRoot(2) });  
}
```

## Approfondimento

### **Incapsulamento (continuazione)**

Sandro Pedrazzini

Incapsolamento (cont) 1

## **Realizzazione 1**

- In una prima realizzazione possibile, lo stato di un oggetto viene rappresentato da variabili di istanza anno, mese, giorno

```
public class Day {  
    private int year;  
    private int month;  
    private int dayOfMonth;  
  
    public Day(int year, int month, int dayOfMonth){  
        this.year = year;  
        this.month = month;  
        this.dayOfMonth = dayOfMonth;  
    }  
  
    public int getDayOfMonth(){  
        return dayOfMonth;  
    }  
    ...
```

Sandro Pedrazzini

Incapsolamento (cont) 2

## Metodi pubblici (1)

- La lettura dello stato con i metodi get è semplice
- Più lunghe sono le operazioni *addDays()* e *daysFrom()*

*Punti da considerare:*

Lunghezza dei mesi

Anni bisestili (divisibili per 4, tranne, dopo il 1582, quelli divisibili per 100, con eccezione dei divisibili per 400)

L'anno zero non esiste, si passa da -1 a +1

Nel passaggio al calendario gregoriano sono saltati 10 giorni: il 15 ottobre 1582 è stato il giorno successivo al 4 ottobre 1582

## Metodi pubblici (2)

- Esempio: *addDays()*

```
public Day addDays(int n) {  
    Day result = new Day(this); //copia  
    while (n > 0){  
        result = result.nextDay();  
        n--;  
    }  
    while (n < 0){  
        result = result.previousDay();  
        n++;  
    }  
  
    return result;  
}
```

## Metodi privati (1)

- Esempi

```
private static int daysPerMonth(int year, int month) {  
    int days = DAYS_PER_MONTH[month - 1];  
    if (month == FEBRUARY && isLeapYear(year)){  
        days++;  
    }  
    return days;  
}  
  
private static boolean isLeapYear(int year) {  
    if (year % 4 != 0){  
        return false;  
    }  
    if (year < GREGORIAN_START_YEAR){  
        return true;  
    }  
    return (year % 100 != 0) || (year % 400 == 0);  
}
```

## Metodi privati (2)

- I metodi ausiliari, come *daysPerMonth()*, *isLeapYear()*, *nextDay()* e *previousDay()*, sono dichiarati *private*.
- Come scegliere se dichiarare un metodo *public* o *private*?
- Con la fatica che si è fatto per realizzarli, perché non metterli a disposizione?

## Metodi privati (3)

- Motivi per essere cauti nel rendere pubblici i metodi ausiliari
  - Possono rendere meno chiara l'interfaccia pubblica (comprensione della classe, coesione)
  - A volte i metodi ausiliari necessitano di un ordine di invocazione specifico.
  - A volte dipendono da una particolare realizzazione della classe. Cambiando realizzazione potrebbero non diventare più necessari. Il fatto di renderli pubblici, obbliga nuove realizzazioni della classe a mettere a disposizione queste funzionalità.
    - » => Una volta resi pubblici, dovranno sempre rimanere pubblici!

## Principi di incapsulamento

- Rendere privati tutti i metodi che non fanno parte dell'interfaccia della classe (non sono di interesse per gli utenti della classe)
- Rendere private tutte le funzionalità che dipendono dall'implementazione della classe.  
=> accoppiamento
- Coesione: che ruolo gioca?

## Variante realizzazione 1

- Day adapter di Calendar o Localdate: delega di compiti a una classe già in grado di realizzarli

```
public class Day{  
    //private Calendar calendar; //fino a Java 8  
    private LocalDate localDate;  
  
    public Day(int year, int month, int dayOfMonth){  
        //calendar = new GregorianCalendar(year, month-1, dayOfMonth);  
        localDate = LocalDate.of(year, month, dayOfMonth);  
    }  
  
    public int getYear(){  
        //return calendar.get(Calendar.YEAR);  
        return localDate.getYear();  
    }  
    ...
```

Sandro Pedrazzini

Incapsolamento (cont) 9

## Variante realizzazione 1 (2)

- Day adapter di Calendar o Localdate.
- **Attenzione:**  
evitare di rendere pubbliche operazioni che creano dipendenze di implementazione.

```
public class Day {  
    ...  
  
    private Day(LocalDate localDate){  
        ...  
    }  
  
    ...  
}
```

Sandro Pedrazzini

Incapsolamento (cont) 10

## Realizzazione 2

- La variante della realizzazione 1 delega a una classe interna la gestione di anni, mesi e giorni, senza modificare la API.
- La prima realizzazione (senza variante) appare inefficiente, perché tutte le funzioni effettuano gli incrementi un giorno alla volta.
- Nuova realizzazione: invece di memorizzare anno, mese e giorno, memorizziamo il numero del giorno juliano, cioè il numero di giorni trascorsi dal 1. gennaio dell'anno 4713 a. C.
- Le funzioni interne devono essere in grado di calcolare il numero del giorno juliano a partire da un giorno del calendario e viceversa

## Giorno Giuliano

- Non ha niente a che vedere con il calendario juliano (di Giulio Cesare, che già prevedeva gli anni bisestili, ma senza la correzione di papa Gergorio sui mutipi di 100 e 400, introdotta nel 1582)
- Si tratta di un punto stabilito nel secolo XVI da Joseph Scaliger, calcolato considerando alcuni eventi astronomici. Lo zero corrisponde al primo gennaio 4713 a. C.)
- Usato come valore iniziale per mettere in corrispondenza in modo affidabile ogni evento della storia scritta con un numero intero positivo che rappresentasse il giorno
- “Giuliano” deriva dal nome del padre di Scaliger, Julius

## Metodi pubblici (1)

- Usando i numeri del giorno juliano, i metodi *addDays()* e *daysFrom()* diventano molto semplici ed efficienti

```
public Day addDays(int n)    {
    return new Day(julian + n);
}

public int daysFrom(Day start)  {
    return julian - start.julian;
}
```

## Metodi pubblici (2)

- I metodi *getYear()*, *getMonth()* e *getDayOfMonth()* devono trasformare l' informazione interna in informazione esterna

```
public Day(int year, int month, int dayOfMonth){
    fJulian = toJulian(year, month, dayOfMonth);
}

public int getYear()    {
    return fromJulianToYear();
}
```

## Metodi privati (1)

- Completamente diversi da quelli precedenti

```
private Day(int julian) {  
    this.julian = julian;  
}  
  
private int fromJulianToYear(){  
    return fromJulian(julian)[0];  
}  
  
private static int[] fromJulian(int julian){  
    ...  
}
```

Sandro Pedrazzini

Incapsolamento (cont) 15

## Metodi privati (2)

```
//Algoritmo da: Press et al., Numerical Recipes in C,  
//                Cambridge University Press 1992  
  
private static int toJulian(int year, int month, int date) {  
    int jy = year;  
    if (year < 0) jy++;  
    int jm = month;  
    if (month > 2) jm++;  
    else {  
        jy--;  
        jm += 13;  
    }  
    int jul = (int) (java.lang.Math.floor(365.25 * jy)  
                    + java.lang.Math.floor(30.6001 * jm) + date + 1720995.0);  
    // 15 ottobre 1582  
    int IGREG = 15 + 31 * (10 + 12 * 1582);  
    // Pasaggio al calendario gregoriano  
    if (date + 31 * (month + 12 * year) >= IGREG) {  
        int ja = (int) (0.01 * jy);  
        jul += 2 - ja + (int) (0.25 * ja);  
    }  
    return jul;  
}
```

Sandro Pedrazzini

Incapsolamento (cont) 16

## Metodi privati (3)

- In questo caso i metodi get sono meno efficienti, perché a loro si chiede continuamente di effettuare conversioni

```
System.out.println(day.getYear() + “.” +
                    day.getMonth() + “.” +
                    day.getDayOfMonth());
```

## Realizzazione 3

Usare entrambe le rappresentazioni interne e passare da una all’ altra in modo “lazy”.

```
public class Day {
    private int year;
    private int month;
    private int dayOfMonth;

    private int julian;

    private boolean ymdValid;
    private boolean julianValid;

    private Day(int julian) {
        this.julian = julian;
        this.ymdValid = false;
        this.julianValid = true;
    }
    ...
```

## Conversioni

```
public Day addDays(int n) {
    ensureJulian();
    return new Day(julian + n);
}

public int getYear() {
    ensureYmd();
    return year;
}

private void ensureYmd() {
    if (!ymdValid){
        int[] ymd = fromJulian(julian);
        year = ymd[0];
        month = ymd[1];
        dayOfMonth = ymd[2];
        ymdValid = true;
    }
}
```

## Incapsulamento (1)

- Anche una classe apparentemente semplice come *Day* può essere realizzata in modi diversi, ognuno con vantaggi e svantaggi
- Usando in modo corretto l'incapsulamento gli utenti della classe *Day* possono rimanere inconsapevoli dei dettagli di implementazione
- Usando in modo corretto l'incapsulamento, i progettisti della classe *Day* possono tranquillamente cambiare la realizzazione, senza creare inconvenienti agli utilizzatori

## Incapsulamento (2)

- Cosa sarebbe successo se nella prima realizzazione avessimo dichiarato *year*, *month* e *dayOfMonth* di tipo *public*, permettendo l'accesso alle variabili?
- Il passaggio dalla realizzazione 1 alla 2 avrebbe obbligato ad introdurre metodi get di sostituzione
- Il passaggio alla realizzazione 3 sarebbe ancora peggio, perché in questo caso avremmo di nuovo accesso alle variabili, di nuovo esistenti, con però semantica leggermente diversa (potrebbero non essere inizializzate). Otterremmo quindi errori in runtime, ma non in compilazione.

## Incapsulamento (3)

- L'incapsulamento costringe a investire più tempo nella progettazione, ma è la chiave per progetti di grosse dimensioni.
- I prodotti software si evolvono nel tempo: si devono aggiungere nuove funzionalità e toglierne altre.
- L'incapsulamento fornisce un meccanismo per ridurre la porzione di programma che viene toccata da un cambiamento.

=> accoppiamento

## Metodi get/set

- Con metodi “get” si intende metodi di accesso alle informazioni interne.
- Con metodi “set” si intende metodi di modifica delle informazioni interne.
- Classi senza metodi “set” (come *Day*) vengono dette *immutable*, perché ogni oggetto creato non può più essere modificato.  
=> la mancanza di set() è uno dei criteri di immutabilità, come vedremo in seguito
- Sia *Date* che *Calendar* contengono invece metodi che ne permettono la modifica

## Problemi con metodi “set” (1)

- Se volessimo aggiungere metodi “set” a Day, dovremmo tener conto della semantica

```
Day deadline = new Day(2019,1,31);
```

Cosa succede con questa operazione?

```
deadline.setMonth(2);
```

Dobbiamo aspettarci che diventi automaticamente un 3 marzo (giorno successivo valido)?

Cosa succederebbe allora in questo caso? 1. marzo invece che 1. febbraio?

```
deadline.setMonth(2);
deadline.setDayOfMonth(1);
```

## Problemi con metodi “set” (2)

Forse basta scambiare l’ordine delle operazioni (un presupposto comunque pericoloso)?

```
Day deadline = new Day(2019,2,1); // 1. febbraio  
  
deadline.setDayOfMonth(30);  
deadline.setMonth(4);
```

Se il metodo setDayOfMonth() avanza fino al giorno valido successivo, allora La scadenza viene prima fissata al 2 marzo, poi al 2 aprile...

## Problemi con metodi “set” (3)

- Non è indispensabile fornire metodi “set” per ogni campo o in coppia con ogni metodo “get”.
- Fornire questi metodi dove è strettamente necessario
- Oggetti senza metodi modificatori possono più facilmente essere condivisi => devono essere immutabili

## Classi immutabili

- Una classe è detta “immutabile” quando le sue istanze non possono essere modificate
- Le informazioni contenute negli oggetti vengono passate durante la creazione e poi sono fisse per tutta la durata di vita dell’oggetto
- La piattaforma Java contiene parecchie classi immutabili:
  - String
  - Classi wrapper dei tipi primitivi (Integer, Float, ecc.)
  - BigInteger, BigDecimal
- Ragioni per definire classi immutabili: più semplici da disegnare, si adattano meglio alla programmazione funzionale, più sicure e robuste, più adatte alla programmazione con thread, ecc.

## Classi immutabili (2)

- **5 regole per rendere una classe immutabile**
  1. Nessun metodo dev’essere in grado di modificare lo stato di un oggetto
  2. Assicurarsi che la classe non possa essere estesa
  3. Definire tutti i campi “final”
  4. Definire tutti i campi privati
  5. Assicurare accesso esclusivo ad ogni componente mutabile

## Regola 1

- Nessun metodo dev'essere in grado di modificare lo stato di un oggetto
  - Questi metodi sono detti metodi "mutator"
  - I metodi set() discussi in precedenza fanno parte di questa categoria di metodi

## Regola 2

- Assicurarsi che la classe non possa essere estesa
  - Questo per impedire che una sottoclasse possa compromettere il comportamento immutabile della classe di base
  - Il comportamento immutabile può essere compromesso anche senza modificare i campi, basta che la sottoclasse "simuli" il comportamento di mutabilità
  - Impedire che una classe venga estesa può essere fatto in Java con il modificatore "final"

## Regola 3

- Definire tutti i campi “final”
  - Dal momento che non ci sono metodi “mutators”, questo provvedimento può sembrare non necessario
  - In questo modo si esprime però il concetto in modo chiaro, attraverso una sintassi capita dal sistema

## Regola 4

- Definire tutti i campi privati
  - Impedisce di avere accesso a oggetti mutabili, a cui i campi si riferiscono. Questi oggetti potrebbero altrimenti essere modificati dal client
  - Sarebbe possibile avere campi pubblici con riferimento a oggetti immutabili, ma in questo caso non si manterrebbe l’incapsulamento

## Regola 5

- Assicurare accesso esclusivo ad ogni componente mutabile
  - Se la classe ha riferimenti a oggetti mutabili, evitare che clienti della classe possano ottenere riferimenti a questi oggetti
  - Mai inizializzare questi campi con semplici riferimenti arrivati dall'esterno o restituire riferimenti diretti come *return* di un metodo di accesso
  - Prevedere copie “difensive” nel costruttore, metodo di accesso e metodo *readObject()* (metodo che legge i dati da serializzazione)

## Esempio di accesso (1)

- Quando invece un oggetto non è immutable, bisogna fare particolare attenzione, perché può violare l'incapsulamento!

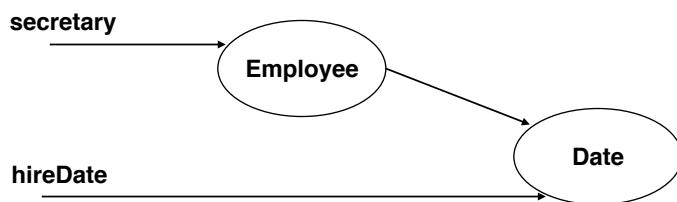
```
public final class Employee {  
    private final String name;  
    private final Date hireDate;  
    ...  
  
    public String getName(){  
        return name;  
    }  
  
    public Date getHireDate(){  
        return hireDate;  
    }  
    ...  
}
```

## Esempio di accesso(2)

- La classe Employee non ha metodi “set”, ma non è da ritenere immutabile
- Il metodo `getHireDate()` viola l’incapsulamento, perché restituisce il riferimento a `Date`, un oggetto modificabile.
- Modificando l’oggetto di `Date`, si modifica l’oggetto impiegato

## Oggetti condivisi (3)

```
Employee secretary = ...  
Date hireDate = secretary.getHireDate();  
hireDate.setTime(t);  
...
```



## Oggetti condivisi (4)

- Il metodo `getHireDate()` era stato progettato per fornire informazioni sull' oggetto che rappresenta un impiegato, non per modificarlo.
- **Possibile soluzione:**

```
public Date getHireDate(){  
    return (Date) hireDate.clone();  
}
```

## Oggetti condivisi (5)

- Il metodo `getName()` non pone problemi, perché un oggetto `String` è immutabile
- La restituzione di un elemento primitivo non pone mai problemi, visto che ne viene sempre fatta una copia
- Bisogna invece fare attenzione agli oggetti di classi modificabili

## Approccio funzionale

- Con classi immutabili, le operazioni sugli oggetti devono seguire l'approccio funzionale, come usato in addDays()

```
public Day addDays(int days)
```

Viene chiamato in questo modo perché la funzione applicata all'operando restituisce un valore senza modificare l'oggetto invoker

## Vantaggi (1)

- **Semplicità**  
Un oggetto immutabile può trovarsi in un unico stato, quello di quando è stato creato
- **Thread-safety**  
Un oggetto immutabile non ha bisogno di sincronizzazione sui thread, perché ha un solo stato possibile

## Vantaggi (2)

- **Condivisione**

Gli oggetti immutabili possono essere condivisi.

La condivisione dovrebbe essere incoraggiata, magari mettendo a disposizione costanti o metodi factory static

- **Esempio**

```
public static final Day CHRISTAMS_2020 = new Day(25, 12, 2020);
```

## Vantaggi (3)

- **Non servono copie “difensive”**

Proprio perché gli oggetti immutabili possono essere liberamente condivisi, non sono necessarie copie “difensive”, come visto nel caso di *String*, rispetto invece a *Date*

In effetti la cosa non è sempre stata chiara nel JDK, in cui è stato previsto un copy constructor per la classe *String*, che in realtà non dovrebbe mai essere necessario usare

## Vantaggi (4)

- **Gli oggetti immutabili non sono solo condivisibili, ma possono condividere tra loro la parte interna**

- **Esempio**

La classe *BigInteger* rappresenta il suo valore con un array di int (magnitude) e il segno con un int (-1 negativo, 1 positivo, 0 per zero, per assicurare rappresentazioni uniche)

Il metodo *negate()* crea e restituisce un nuovo oggetto *BigInteger*, con segno contrario, ma con array interno condiviso.

## Vantaggi (5)

- **Gli oggetti immutabili sono le migliori componenti per altri oggetti (siano questi a loro volta immutabili o mutabili)**

- È molto più semplice mantenere le invarianti di un oggetto complesso, sapendo che le sue componenti non variano

- Gli oggetti immutabili sono i migliori elementi per essere usati come chiavi in Map o Set: non cambiano nel tempo, distruggendo di fatto le invarianti delle tabelle

## Svantaggio

- **Necessario un nuovo oggetto per ogni nuovo valore**

- Questo può essere costoso se gli oggetti sono grossi (modifica di un singolo bit in un *BigInteger* di milioni di elementi)
- Può essere costoso se un'operazione su uno di questi oggetti necessita più passi, in ognuno dei quali viene creato un nuovo oggetto che poi viene gettato.

In questi casi si può gestire i vari passaggi con classi intermedie mutabili, sia private o “package” per una singola classe, sia pubbliche, come nel caso di *StringBuilder* (o *StringBuffer*) per le operazioni da evitare con *String*.

=> Queste classi sono dette “mutable companion” di *String*

## Separazione accesso/modifica (1)

- Anche se è meglio evitarli quando non sono indispensabili, i metodi modificatori sono importanti, perché permettono ad un oggetto di cambiare il suo stato nel tempo.
- Quando abbiamo sia metodi di accesso che metodi di modifica, è bene cercare di tenerne ben separati i ruoli
- In una classe *ContoBancario* non ci si aspetta certo che il metodo *leggiBilancio()* vada a modificare il saldo...

## Separazione accesso/modifica (2)

- Ci si aspetta che l' utilizzo di un metodo di accesso possa essere eseguito più volte, ottenendo sempre il medesimo risultato
- La classe  *StringTokenizer* ha un metodo (*nextToken()*), che viola questa regola: restituisce il valore successivo della stringa che si sta scomponendo

```
StringTokenizer tokenizer = new StringTokenizer(content, " .");
while(tokenizer.hasMoreTokens()){
    String str = tokenizer.nextToken();
    ...
}
```

## Separazione accesso/modifica (3)

- Come separare le due funzioni? Usando due metodi

- **String getToken() //lettura del token attuale**
- **void nextToken() //spostamento al token successivo**

```
StringTokenizer tokenizer = new StringTokenizer(content, " .");
while(tokenizer.hasMoreTokens()){
    tokenizer.nextToken();
    String str = tokenizer.getToken();
    ...
}
```

## Separazione accesso/modifica (4)

- Il nuovo metodo `getToken()` può essere chiamato più volte, senza modificare lo stato dell'oggetto.
- Se in alcuni casi dà fastidio dover eseguire due chiamate per ottenere l'elemento successivo (come nel nostro caso: `nextToken() + getToken()`), bisognerebbe almeno mettere sempre a disposizione un metodo (`getToken()` nel nostro caso) per un accesso senza modifiche

## Effetti collaterali (1)

- Un metodo di accesso non modifica lo stato dell' oggetto invocante (parametro隐式的), mentre un metodo modificatore lo fa.
- Qualsiasi altra modifica apportata da un metodo viene chiamata effetto collaterale.

## Effetti collaterali (2)

- Ci si aspetta che i metodi di una classe non modifichino i parametri esplicativi

```
lista.addAll(sottolista)
```

- Con la chiamata ad `addAll()` intendiamo aggiungere una sottolista (parametro esplicito) ad una lista (oggetto invocante)
- Ci aspettiamo che *lista* venga modificata, ma NON *sottolista*

## Accesso ai parametri

- Quando una classe utilizza altre classi al suo interno per gestire i suoi dati, ci si aspetta che queste classi non vengano “viste” (e quindi utilizzate) all’ esterno della classe.
- Quando queste vengono utilizzate anche all’ esterno (per necessità, per scelta di design), bisogna essere consapevoli che non è più possibile organizzare i dati in altro modo

## Approfondimento

### **Incapsulamento (introduzione)**

## **Motivazione**

**Importanza dell' incapsulamento nella programmazione OO**

**Ruolo degli elementi “interface”**

**Qualità delle classi e delle interfacce**

**Non solo le relazioni tra classi sono importanti nello sviluppo,  
ma anche la buona manifattura di ogni singola classe**

**Classi utili e riutilizzabili**

## Classe Date

- Molti programmi hanno la necessità di elaborare informazioni relative alla data
- In Java esiste la classe `java.util.Date`

```
Date date = new Date();
System.out.println(date);
```

## Metodi di Date

```
boolean after(Date date);
boolean before(date date);
int compareTo(Date anotherDate);
long getTime(); //millisecondi da 1-1-1970
void setTime();
```

## Esempio

In realtà, da quando è stata inserita la classe *Calendar*, si tratta più di una classe *Time* (istante di tempo) che di una classe *Date*.

```
Date date = Calendar.getInstance().getTime();

date.setTime(0);
System.out.println(date);      //01-01-1970

date.setTime(-864000000);     //valore "long"
System.out.println(date);      //22-12-1969

date.setTime(864000000);
System.out.println(date);      //11-01-1970
```

## Ordinamento totale

- Con due oggetti di tipo *Date* è possibile chiedersi quale dei due preceda l'altro
- Gli istanti di tempo sono dotati di ordinamento totale, implementati dai metodi *after()* e *before()*

```
t1.after(t2)  
corrisponde a  
t1.getTime() > t2.getTime()
```

## Altre informazioni (1)

- Date non permette però di ottenere informazioni relative al mese, anno di appartenenza, ecc. (in realtà lo permetteva, ma i metodi sono ora dichiarati “deprecated”).
- La responsabilità di determinare queste informazioni è affidata alla classe *GregorianCalendar* (calendario introdotto da Papa Gregorio XIII nel 1582 con la correzione dell'algoritmo sugli anni bisestili, già comunque presenti nella forma più semplice nel calendario giuliano derivante da Giulio Cesare).
- La classe *GregorianCalendar* conosce i dettagli e le complicazioni (30 giorni ha novembre, ...) dei nostri calendari.

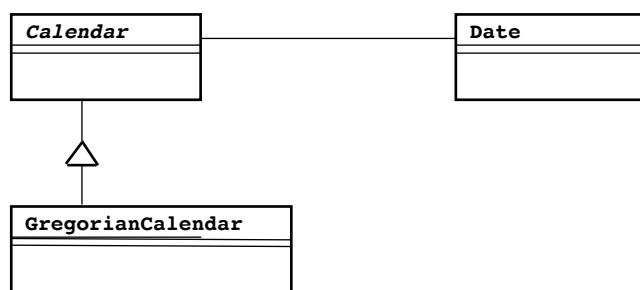
## Altre informazioni (2)

- Il fatto di separare le informazioni di *GregorianCalendar* da *Date* (che rappresenta in effetti unicamente un istante di tempo) è un buon esempio di progettazione, perché esistono molte descrizioni di tempo:
  - Calendario Gregoriano: 3 febbraio 2001
  - Calendario della Rivoluzione francese: Année 209 de la République, Mois de Pluviose, 2 Jour du Quintidi
  - Calendario Maya: 12.19.7.17.1

## Responsabilità

- La responsabilità che spetta a *GregorianCalendar* è quella di assegnare descrizioni agli istanti di tempo.
- Un istante di tempo è “neutro”, mentre la sua descrizione può essere fatta in vari modi (ad esempio attraverso una nuova classe *MayanCalendar*).
- È questo il motivo per cui, almeno nelle intenzioni, è utile avere una classe astratta *Calendar*, da cui classi concrete come *GregorianCalendar* possano ereditare.

## Struttura



## Metodi di *Calendar*

- *Calendar* specifica una serie di metodi “generici” (non legati a un calendario specifico)

```
...
public int get(int field)
public void set(int field, int value)
public void add(int field, int increment)

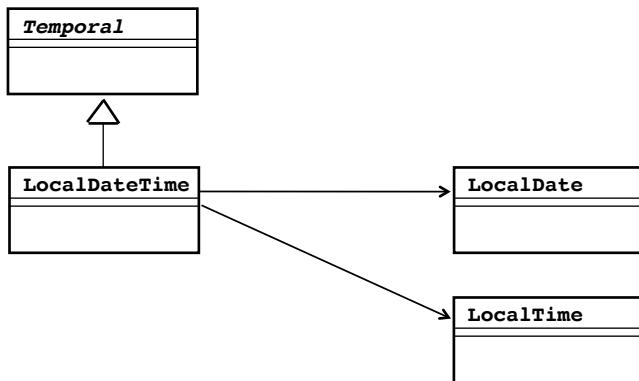
public Date getTime()
public void setTime(Date date)
```

## LocalDate e LocalTime

- Classi definite in Java 8, thread-safe.
- Sono “local”, perché rappresentano data e ora dal punto di vista dell’osservatore.
- Qui “data” torna ad essere un oggetto con informazioni e operazioni su data (calendario), mentre LocalTime rappresenta l’orario.
- LocalDateTime rappresenta entrambi.

```
LocalDateTime timePoint = LocalDateTime.now(); //ora e data attuali
LocalDate.of(2017, Month.DECEMBER, 12); // data dai valori
LocalTime.of(17, 45); // data dai valori 17:45
LocalTime.parse("10:15:30"); // From a String
```

## Struttura



## Metodi di *Temporal*

- *Temporal* specifica una serie di metodi “generici” (non legati a un calendario specifico, implementato da *LocalDate*, *JapaneseDate*, *ThaiBuddhistDate*, ecc.)
- *TemporalUnit* specifica ore o giorni
- *TemporalField* specifica il tipo di campo (giorno della settimana, giorno dell’anno, ecc.)

...

```
Temporal with(TemporalField field, long newValue);
Temporal plus(long amountToAdd, TemporalUnit unit);
Temporal minus(long amountToAdd, TemporalUnit unit);
long until(Temporal endExclusive, TemporalUnit unit);
```

## Realizzazione: classe Day

- Realizzazione di una classe *Day* del calendario gregoriano
- Un oggetto di tipo *Day* rappresenta un giorno
- Evitiamo di rendere esplicativi dettagli di implementazione, come "inizio dei tempi" (*epoch*, 1 gennaio 1970), utilizzato da *Date*
- Mette a disposizione metodi di utilità quali:
  - Quanto manca alla fine dell' anno?
  - Quale giorno sarà tra 50 giorni?
  - Quanti giorni ci sono dalla data X alla data Y?

## Operazioni

```
public int daysFrom(Day start)
    // numero di giorni da start ad ora

public Day addDays(int days)
    // aggiunta di un numero di giorni
```

## Utilizzo

```
public static void main(String[] args){
    Day today = Day.now();
    Day later = today.addDays(999);

    System.out.println(later.getYear()
        + " - " + later.getMonth()
        + " - " + later.getDate());

    System.out.println(later.daysFrom(today)); // 999
}
```

## Realizzazione 1

- In una prima realizzazione possibile, lo stato di un oggetto viene rappresentato da variabili di istanza anno, mese, giorno

```
public class Day{
    private int year;
    private int month;
    private int dayOfMonth;

    public Day(int year, int month, int dayOfMonth){
        this.year = year;
        this.month = month;
        this.dayOfMonth = dayOfMonth;
    }

    public int getDayOfMonth(){
        return dayOfMonth;
    }
    ...
}
```

# Ingegneria del Software: Incapsulamento e API

## Descrizione del problema

Sfruttando le astrazioni disponibili in Java per la gestione del tempo (Calendar, LocalDate da Java8), realizzare una classe Day immutabile, capace di gestire un istante di tempo in giorni, con possibili operazioni su di essa.

## Compiti

Specificare alcune informazioni pubbliche della classe *Day* (*getYear()*, *getMonth()*, ecc.) e alcune funzionalità di API, che siano in grado di rispondere a richieste che vanno oltre a quelle di ordinamento totale (prima, dopo).

### Esempi:

- Quanti giorni mancano da oggi alla fine dell'anno?
- Quale giorno sarà tra 50 giorni a partire da oggi?
- Quanti giorni ci sono dalla data X alla data Y?

Dopo aver determinato le funzionalità di API, passare alla struttura interna delle informazioni da gestire e alla realizzazione completa della classe.

## Procedimento

Determinare la struttura di dati da gestire all'interno della classe.  
Realizzare la classe e alcuni esempi che la utilizzino.

### Punti generali da verificare:

- Cosa va definito pubblico e cosa privato?
- È possibile cambiare la rappresentazione interna della classe *Day*, senza modificarne la API?

## Qualità nei processi di sviluppo

### Integrazione continua

### Integrazione continua

- Pratica dello sviluppo software
- Ogni membro del gruppo di sviluppo integra il proprio lavoro il più frequentemente possibile, solitamente almeno una volta al giorno
- Ogni integrazione viene verificata da un *build* automatico, che lancia i test e verifica eventuali problemi di integrazione

## Integrazione continua (2)

- Molti team di sviluppo trovano che una pratica del genere serve a ridurre al minimo i problemi di integrazione
- Permette al team di sviluppare software coeso in modo molto più rapido
- Aiuta a ridurre i rischi legati allo sviluppo e all'integrazione

## Motivazione

- In molti processi software l'integrazione richiede giorni, settimane o mesi
- Questo perché l'integrazione viene eseguita come ultimo passaggio prima di andare in produzione
- Per poter andare in produzione più frequentemente bisogna trasformare l'integrazione in qualcosa di automatico (integrazione vista come “non-event”)

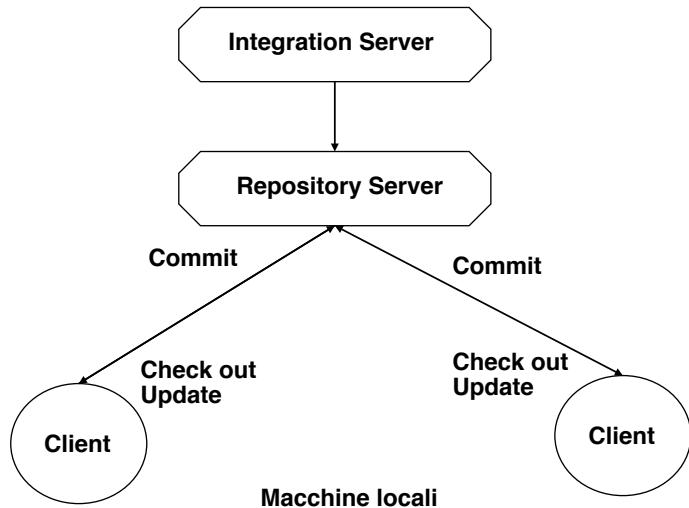
## Motivazione (2)

- Integrando regolarmente significa che ogni sviluppatore ad ogni iterazione aggiunge solo alcune ore di lavoro al progetto integrato
- Anche in questo caso (come nelle altre pratiche di XP) si tratta semplicemente di applicare in modo coerente alcune “buone abitudini”.
- Più frequentemente si integra e più l’integrazione diventa un “non-event”

## Tool

- Integrazione continua è prima di tutto una buona abitudine, perciò di base non necessitano tool particolari
- Ne esistono però alcuni (esempi: **CruiseControl**, **TeamCity**, **Jenkins**) che combinati a un sistema di gestione concorrente di sorgente (**CVS**, **Subversion**, **GIT**, ecc.) e a un sistema di build indipendente da ogni IDE (esempio: Ant, Maven, Gradle) permettono di attivare le verifiche di integrazione in modo automatico.

## Tool (2)

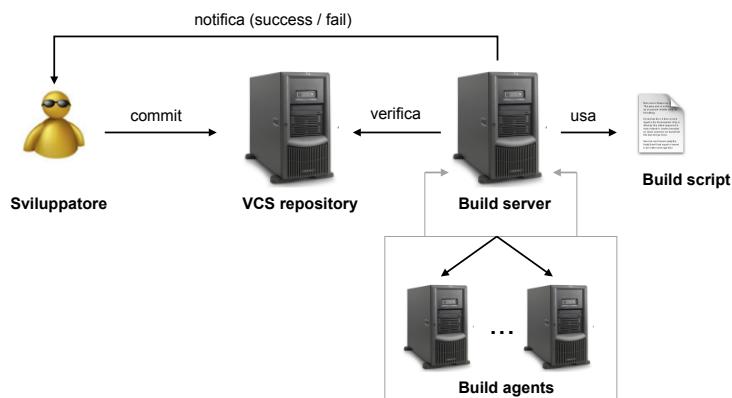


Sandro Pedrazzini

Integrazione continua

7

## Processo



Sandro Pedrazzini

Integrazione continua

8

## Tool (3)

- Esempio di build con Ant

➤ ant integrate

```
Buildfile: build.xml
clean:
all:
compile-src:
compile-tests:
integrate-db:
run-tests:
run-inspections:
delivery:
deploy:
BUILD SUCCESSFUL
Total time: 6 minutes 15 seconds
```

## Tool (4)

- Esempio di commit con Subversion (via linea di comando)

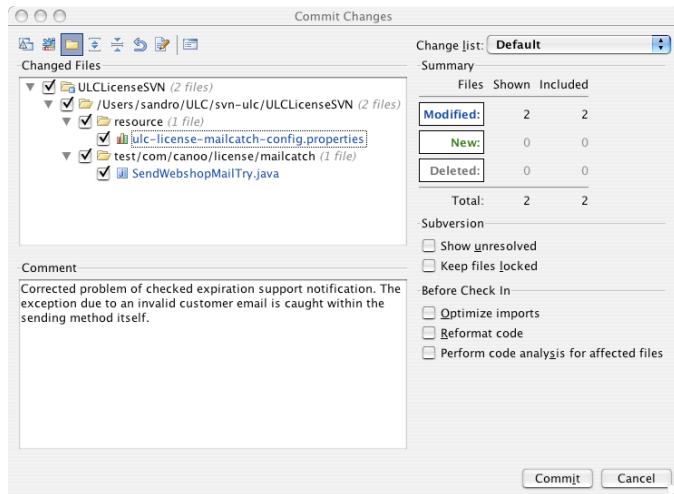
➤ svn commit -m "Aggiunta verifica della connessione"

```
Sending src/com/canoo/db/dao/DBServicesImpl.java
Transmitting file data .

Committed revision 125
```

## Tool (5)

- Esempio di commit integrato in IDE



Sandro Pedrazzini

## Integrazione continua

11

## Tool (6)

- **Confronto fra versioni**

```
Ignore whitespace: Do not ignore
```

```
10040(Read-only) 10351
    } else {
        mailContent.append("No new expired support license");
    }
    mailContent.append("Expired support licenses: " + checkedSupportLicenses.size() + "\n");
    fillMailContent(mailContent, expirationBuckets);
    MailHandler.sendSimpleMessage(baseConfig.EMAIL_ADMIN, null, null, baseConfig.EMAIL_EXP_SI);
    if (baseConfig.isEmailNotificationForSingleClients) {
        sendEmailNotificationForSingleClients(bucketsOfExpiredLicenses);
    }
    return (SupportLicense[][]) checkedSupportLicenses.toArray(new SupportLicense[checkedSupportLicenses.size()]);
}

private static void sendEmailNotificationForSingleClients(Map expirationBuckets) throws Exception {
    if (expirationBuckets.size() == 1) {
        String singleClientEmail = expirationBuckets.keySet().iterator().next();
        String subject = "Send notification to each single client";
        String message = "Dear " + singleClientEmail + ",\n" +
            "Your single client license has expired.\n" +
            "Please contact us for further information.\n";
        Iterator<String> iterator = sortedKeys.iterator();
        while (iterator.hasNext()) {
            String expirationBucketKey = iterator.next();
            Map<String, SupportLicense> expirationBucket = expirationBuckets.get(expirationBucketKey);
            ProductAndSupportLicensePair licenseFair = (ProductAndSupportLicensePair) expirationBucket.get(singleClientEmail);
            ContentList contentList = licenseFair.getLicenseContentList();
            String content = contentList.get(licenseFair.getLicenseContentIndex());
            MailHandler.sendSimpleMessage(expirationBucketKey, getRecipientEmail(), prepareResellerEmail(content));
        }
    }
}

private static String prepareResellerAddress(String secondContactEmail) {
    if (baseConfig.RESELLER_AC_CC != null && !secondContactEmail.equals("")) {
        return secondContactEmail;
    }
    return null;
}

private static String prepareCustomerText(ProductAndSupportLicenseFair licenseFair, List contentList) {
    StringBuilder text = new StringBuilder();
    text.append("Dear " + licenseFair.getLicenseeName() + ",\n" + append(licenseFair.getLicenseContentList().get(licenseFair.getLicenseContentIndex())));
    text.append(baseConfig.CUSTOMER_MAIL_TEXT1);
    text.append("\n");
    while (item.hasNextItem()) {
        ProductAndSupportLicenseFair pair = (ProductAndSupportLicenseFair) item.next();
        text.append(pair.getLicenseContentList().get(licenseFair.getLicenseContentIndex()));
    }
    text.append("\n");
    text.append(baseConfig.CUSTOMER_MAIL_TEXT2);
    return text.toString();
}

private void fillBucketWithRelatedExpirationLicenses(Map expirationBucket, SupportLicense[] supportLicenses) {
    for (SupportLicense license : supportLicenses) {
        Client client = relatedProductLicenses[license.getBucketIndex()];
        String key = client.getEmail() + " " + client.getCompanyName() + " " + supportLicense.getExpirationCode();
        if (client != null) {
            singleLicense.getBucket().put(key, license);
        }
        key = key + client.getEmail() + " " + client.getCompanyName(); // singleLicense.getBucket()
    }
    List<ContentList> contentList = (List<ContentList>) expirationBucket.getKey();
    if (contentList == null) {
        expirationBucket.setKey(contentList);
    }
}

private static void sendEmailNotificationForSingleClients(Map expirationBuckets) throws Exception {
    if (expirationBuckets.size() == 1) {
        String singleClientEmail = expirationBuckets.keySet().iterator().next();
        String subject = "Send notification to each single client";
        String message = "Dear " + singleClientEmail + ",\n" +
            "Your single client license has expired.\n" +
            "Please contact us for further information.\n";
        Iterator<String> iterator = sortedKeys.iterator();
        while (iterator.hasNext()) {
            String expirationBucketKey = iterator.next();
            Map<String, SupportLicense> expirationBucket = expirationBuckets.get(expirationBucketKey);
            ProductAndSupportLicensePair licenseFair = (ProductAndSupportLicensePair) expirationBucket.get(singleClientEmail);
            ContentList contentList = licenseFair.getLicenseContentList();
            String content = contentList.get(licenseFair.getLicenseContentIndex());
            MailHandler.sendSimpleMessage(expirationBucketKey, getRecipientEmail(), prepareResellerEmail(content));
        }
    }
}

private static void sendEmailNotificationForSingleClients(Map expirationBuckets) throws Exception {
    if (expirationBuckets.size() == 1) {
        String singleClientEmail = expirationBuckets.keySet().iterator().next();
        String subject = "Send notification to each single client";
        String message = "Dear " + singleClientEmail + ",\n" +
            "Your single client license has expired.\n" +
            "Please contact us for further information.\n";
        Iterator<String> iterator = sortedKeys.iterator();
        while (iterator.hasNext()) {
            String expirationBucketKey = iterator.next();
            Map<String, SupportLicense> expirationBucket = expirationBuckets.get(expirationBucketKey);
            ProductAndSupportLicensePair licenseFair = (ProductAndSupportLicensePair) expirationBucket.get(singleClientEmail);
            ContentList contentList = licenseFair.getLicenseContentList();
            String content = contentList.get(licenseFair.getLicenseContentIndex());
            MailHandler.sendSimpleMessage(expirationBucketKey, getRecipientEmail(), prepareResellerEmail(content));
        }
    }
}

private static String prepareResellerAddress(String secondContactEmail) {
    if (baseConfig.RESELLER_AC_CC != null && !secondContactEmail.equals("")) {
        return secondContactEmail;
    }
    return null;
}

private static String prepareCustomerText(ProductAndSupportLicenseFair licenseFair, List contentList) {
    StringBuilder text = new StringBuilder();
    if (baseConfig.CUSTOMER_MAIL_AC_CC != null && !secondContactEmail.equals("")) {
        text.append("Dear " + licenseFair.getLicenseeName() + ",\n" + append(licenseFair.getLicenseContentList().get(licenseFair.getLicenseContentIndex())));
    }
    text.append(baseConfig.CUSTOMER_MAIL_TEXT1);
    text.append("\n");
    while (item.hasNextItem()) {
        ProductAndSupportLicenseFair pair = (ProductAndSupportLicenseFair) item.next();
        text.append(pair.getLicenseContentList().get(licenseFair.getLicenseContentIndex()));
    }
    text.append("\n");
    text.append(baseConfig.CUSTOMER_MAIL_TEXT2);
    return text.toString();
}
```

Sandro Pedrazzini

Integrazione continua

12

## Tool (8)

- Mail ricevuta da Cruise Control dopo il build

### BUILD COMPLETE - build.180

Date of build: 01/07/2008 16:21:50  
Time to build: 10 seconds  
Last changed: 01/07/2008 16:16:51  
Last log entry: Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Modifications since last successful build: (6)

modified sandro /ULCLicense/trunk/src/com/canoo/license/base/data/transaction/TransactionContext.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
added sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification-3.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/src/com/canoo/license/base/support/CheckExpiringSupport.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/build.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicenseTemplates/trunk/ULCBase/ulc-support-additions.sql	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Sandro Pedrazzini

Integrazione continua

13

## Tool (9)

### Progetti in TeamCity

Welcome, sandro Logout

Projects My Changes Agents (1) Build Queue (0) Administration My Settings & Tools Configure Visible Projects

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- **CobraWunelli Maintenance Build Configuration** Idle Run
- #build.563-2 Tests passed: 159
- No artifacts Changes (1) 10 Feb 17:56 (7m:37s)
- **CobraWunelli Trunk Build Configuration** Idle Pending (1) Run
- #build.680 Tests passed: 191
- No artifacts Changes (1) 25 Feb 15:33 (12m:09s)

Sandro Pedrazzini

Integrazione continua

14

## Tool (10)

- History di TeamCity

The screenshot shows the TeamCity web interface for the project 'CobraWunelli'. The top navigation bar includes 'Projects', 'My Changes', 'Agents (1)', 'Build Queue (0)', 'Administration', 'My Settings & Tools', and a search bar. The main content area displays the 'History' tab of the build configuration 'CobraWunelli Trunk Build Configuration'. It shows a table of recent builds with columns for '#', 'Results', 'Artifacts', 'Changes', 'Started', 'Duration', 'Agent', 'Tags', and a 'Pin' icon. The table lists builds from #build.680 to #build.666. Build #680 is green (Tests passed: 191). Builds #678, #677, #676, #672, #671, #670, and #669 are green (Tests passed: 185, 172, 172, 175, 175, 175, 175). Build #668 is red (Tests failed: 1 (1 new), passed: 171). Build #666 is red ([Execution timeout] Tests passed: 171). The status bar at the bottom indicates 'Integrazione continua' and the page number '15'.

## Esempio (1)

- Supponiamo di voler aggiungere una nuova funzionalità ad un programma esistente

- Come primo passaggio devo scaricare la versione più aggiornata sulla mia macchina locale (check-out o update, nel caso avessi già una versione locale non aggiornata)
- Quando ho la versione sulla mia macchina, posso iniziare a fare le aggiunte desiderate
- Questo significa non solo aggiungere funzionalità, ma anche adattare vecchio codice, aggiungere e adattare test

## Esempio (2)

- Appena terminato con la nuova funzionalità e i vari test, posso creare il build sulla mia macchina locale
- Questo significa ricompilare tutto, creare e includere le varie librerie, eseguire i test
- Posso considerare di aver terminato il lavoro sulla macchina locale solo quando tutto funziona senza errori
- Ora che il build locale funziona, posso pensare di eseguire un “commit” sul repository comune

## Esempio (3)

- Non ho ancora finito: a questo punto si tratta di eseguire il build sulla macchina di integrazione
- Solo se anche questo build ha successo, si può affermare che i cambiamenti eseguiti fanno parte dell’applicazione
- Il build di integrazione può essere eseguito manualmente oppure automaticamente con tool come TeamCity, che, eseguendo un polling continuo sul repository per determinare se ci sono stati aggiornamenti, fa partire il build quando necessario

## Conflitti (1)

- Se ci sono conflitti a causa di due aggiornamenti che si accavallano, solitamente la cosa viene notata dal secondo che esegue commit
- Significa che dal suo ultimo update, qualcun altro ha eseguito un commit
- Il caso viene risolto localmente con un nuovo update (ed eventuali adattamenti) prima del commit

## Conflitti (2)

- Se il conflitto non si manifesta durante il commit (perché non sono stati toccati gli stessi file), ma esiste comunque una inconsistenza, questa viene scoperta durante il build di integrazione
- In entrambi i casi il problema viene scoperto velocemente
- La cosa più urgente da fare in questi casi è riparare il build

## Conflitti (3)

- In un ambiente di integrazione continua non si dovrebbe mai lasciare un “failed build” troppo a lungo
- Un buon team dovrebbe avere più di un build corretto al giorno
- Ognuno può quindi sviluppare a partire dall’ultima versione del build, mantenendo quindi il delta tra sviluppo e successiva integrazione il più piccolo possibile

## Elementi di CI

- Quanto visto nell’esempio precedente dimostra CI (continuous integration) nel lavoro di tutti i giorni
- Vediamo quali sono gli elementi essenziali affinché tutto questo possa avvenire in modo naturale e senza grossi problemi

## Elemento 1: Un solo repository

- I progetto software richiedono la gestione di parecchi file: utilizzare un sistema di versioning control
- Fare in modo che sia accessibile a tutti, anche da remoto
- Nel repository dovrebbe esserci tutto quanto server per eseguire il build, inclusi script, property file, librerie, ecc.
- Regola: dev'essere possibile eseguire un check out su una macchina "verGINE" ed poter subito eseguire un build

## Elemento 2: Build automatico (1)

- Build significa trasformare i sorgenti in un sistema funzionante
- Questo può essere un processo complicato che include compilazione, spostamento di file, caricamento di uno schema di DB, ecc.
- Tutto questo può essere automatizzato ed è buona cosa che lo sia, perché fa risparmiare parecchio tempo

## Elemento 2: Build automatico (2)

- Ambienti di build automatico ne esistono parecchi: Make in Unix, Ant, Maven, Gradle nel mondo Java, Nant o MSBuild per .NET, ecc.
- A dipendenza delle necessità si deve poter creare build in modo condizionale, avendo a disposizione diversi target (build con codice di test integrato, con verifiche diverse, ecc.)
- Il build può essere creato via IDE, ma dev'essere in ogni caso possibile creare un master sul server, senza IDE

## Elemento 3: Build self-testing

- Build non significa solo compilazione e link, un programma compilato correttamente può avere errori di esecuzione
- Il modo migliore per verificare il funzionamento è inserire la chiamata ai test nel processo di build
- Se un test non passa, il build deve fallire

## **Elemento 4: Si integra ogni giorno (1)**

- L'integrazione è anche un mezzo per comunicare agli altri sviluppatori quali modifiche abbiamo apportato al codice
- Integrando regolarmente si scoprono prima eventuali conflitti di sincronizzazione tra sviluppatori e si possono correggere velocemente
- Conflitti che rimangono irrisolti per giorni o settimane, sono difficili da riparare

## **Elemento 4: Si integra ogni giorno (2)**

- Regola: ogni sviluppatore dovrebbe eseguire un commit almeno una volta al giorno
- Commit frequenti (anche più volte al giorno) incoraggiano lo sviluppatore a suddividere il suo lavoro in fasi di alcune ore l'una. Questo permette di “sentire” il progredire del progetto

## Elemento 5: Mantenere il build veloce (1)

- Un elemento essenziale dell'integrazione continua è il feedback veloce
- Le “guidelines” di eXtreme Programming parlano di un massimo di 10 minuti
- Molto spesso se c’è un problema a mantenere il build a 10 minuti, questo è provocato dai test, soprattutto quelli che fanno uso di servizi esterni, come DB

## Elemento 5: Mantenere il build veloce (2)

- In caso di build troppo lunghi, si deve fare in modo di organizzare il processo a tappe (*staged build*, o *build pipeline*)
- Si parte da un “commit build” che deve durare al massimo 10 minuti, poi si possono organizzare passaggi successivi.
- Il commit build viene usato come punto di riferimento per il ciclo di integrazione continua

## Elemento 5: Mantenere il build veloce (3)

- **Esempio: two stage build**

- Il commit build si occupa della compilazione e dell'esecuzione dei test più essenziali, utilizzando oggetti "Mock" per i test sul DB
- Una volta eseguito questo (nei 10 minuti massimi) esiste un build non affidabile al 100%, ma sufficientemente sicuro da poter permettere agli sviluppatori di basare le proprie modifiche su questo ultimo build
- La seconda fase prevede l'utilizzo di tutti i test. A questo punto può anche durare alcune ore
- Se nella seconda fase si riscontrano problemi, si cerca di creare nuovi test per la prima fase che permettano di scoprire in anticipo quanto si è trovato di problematico

## Elemento 6: clonare l'ambiente di produzione (1)

- È importante poter eseguire tutti i test in un ambiente il più possibile simile a quello di produzione
- Ogni differenza può essere motivo di errore in produzione
- Clonare significa avere le stesse versioni del software, del sistema operativo, del DB, stesse librerie, stesso HW
- Ci sono dei limiti (HW troppo caro), ma spesso i vantaggi li superano ampiamente

## **Elemento 6: clonare l'ambiente di produzione (2)**

- Quando l'applicazione deve poter andare in produzione su ambienti diversi (esempio: applicazione desktop) è quasi impossibile provare tutte le combinazioni
- In questi casi possono però aiutare gli ambienti virtuali (VMWare, Parallels, ecc.)

## **Elemento 7: Rendere accessibile il build (1)**

- Uno dei maggiori problemi nello sviluppo del software è sapere se si stanno sviluppando le funzionalità realmente desiderate dal committente
- Le persone trovano più semplice vedere qualcosa di incompleto, ma che permetta di capire meglio cosa si desidera e come esprimere
- I processi agili si basano su questo e traggono vantaggio da questo tipico comportamento umano

## **Elemento 7: Rendere accessibile il build (2)**

- In queste situazioni è importante che ogni sviluppatore abbia facile accesso all'ultimo build, o al build della sua ultima iterazione
- Lo scopo è quello di poterlo usare per dimostrazioni, test, o anche unicamente per verificare cosa è cambiato negli ultimi giorni
- Il repository deve essere organizzato in modo che ci sia una chiara sequenza storica dei build

## **Elemento 8: Visione dello stato (1)**

- CI serve anche alla comunicazione, perciò è importante che ognuno possa verificare lo stato del sistema e i cambiamenti effettuati
- I programmi di CI hanno un'interfaccia Web che permette di accedere alla storia dei cambiamenti, sapere chi ha fatto le modifiche, ecc.
- Il vantaggio di un'interfaccia Web consiste nel fatto che anche sviluppatori localizzati altrove hanno facile accesso alle informazioni

## Elemento 8: Visione dello stato (2)

- Interfaccia Web di TeamCity

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- = **CobraWunelli Maintenance Build Configuration**  
#build.563-2 Tests passed: 159 | No artifacts Changes (1) 10 Feb 17:56 (7m:37s)  
Idle Run
- = **CobraWunelli Trunk Build Configuration**  
#build.680 Tests passed: 191 | No artifacts Changes (1) 25 Feb 15:33 (12m:09s)  
Idle Pending (1) Run

Sandro Pedrazzini Integrazione continua 37

## Elemento 9: Deployment automatico

- Un elemento di integrazione continua è il deployment automatico
- Questo può essere utile soprattutto se si hanno diversi deployment in ambienti diversi: automatizzare significa evitare facili fonti di errore
- Se è compresa la funzionalità di deployment in produzione, una capacità interessante è quella del rollback automatico alla versione precedente: questo permette di eliminare la “tensione” tipica del deployment

## Benefici della CI (1)

- Minor rischio
  - Ricordo di progetti senza integrazione continua: progetto terminato, ma incognita dell'integrazione
  - Difficile prevedere il tempo necessario di un'integrazione prevista solo alla fine del progetto
  - CI permette di sapere in ogni momento a che punto si è con il progetto e con gli errori

## Benefici della CI (2)

- Errori
  - CI non ci permette di eliminare gli errori, ma ci rende più semplice il compito di trovarli
  - Quando si introduce un bug nel sistema, se lo si scopre in fretta, diventa semplice anche toglierlo
  - Inoltre l'errore può essere solo introdotto in poche ore di lavoro dall'ultimo build stabile
  - Si eliminano i bug accumulati nel tempo

## Benefici della CI (3)

- Deployment più frequente
  - Grossa barriera che CI permette di eliminare
  - Permette di mostrare più rapidamente nuove features all'utente finale e ottenere di conseguenza un feedback più veloce
  - Attraverso feedback più veloce, utente e sviluppatore migliorano il loro rapporto di collaborazione
  - Si accorcia la distanza che esiste tipicamente tra sviluppatore e utente, decisiva per uno sviluppo software di successo

## Integrazione con le altre pratiche (1)

- L'utilizzo di CI si integra bene con altre pratiche di progettazione e sviluppo trattate
  - Test di unità
  - Utilizzo di standard nel codice
  - Refactoring
  - Cicli di sviluppo corti
  - Appartenenza comune del codice

## Integrazione con le altre pratiche (2)

- **Test di unità**

- Chi sviluppa, dovrebbe aggiungere codice di test al proprio codice (unit testing)
- Il test dovrebbe essere richiamato dopo ogni cambiamento
- Con CI il test viene richiamato automaticamente ad ogni build, quindi ad ogni modifica nel repository (regression test)

## Integrazione con le altre pratiche (3)

- **Utilizzo di standard**

- In ogni progetto si definiscono delle *guidelines* da seguire, in modo che l'intero codice segua gli stessi "standard"
- Spesso il controllo dell'aderenza allo standard è un processo manuale
- Con CI si può inserire una serie di analisi statiche del codice nello script di build, in grado di generare un report

## Integrazione con le altre pratiche (4)

- **Refactoring**

- Refactoring significa adattare il codice e la sua struttura interna senza modificare la funzionalità
- Uno degli scopi è quello di facilitare la manutenzione del codice
- CI assiste lo sviluppatore permettendo la chiamata di tool di inspection del codice all'interno del build, eseguito ad ogni modifica del repository

## Integrazione con le altre pratiche (5)

- **Cicli brevi**

- Significa che gli utenti devono avere a disposizione l'ultima versione del software funzionante il più spesso possibile
- CI si adatta bene a questa pratica, perché si integra più volte al giorno e ogni integrazione genera virtualmente un nuovo release
- Quando un sistema di integrazione continua è installato, un nuovo release viene generato con il minimo degli sforzi

## Integrazione con le altre pratiche (6)

- **Appartenenza comune (*collective ownership*)**
  - Ogni sviluppatore può lavorare a qualsiasi parte del sistema
  - Questo impedisce che ci sia un solo sviluppatore con conoscenze specifiche di un determinato argomento
  - CI aiuta questa pratica assicurando aderenza agli standard e richiamando continuamente i test di regressione

## Ridurre i rischi (1)

- **CI aiuta nel mantenere i rischi sotto controllo, permettendo di scoprire i problemi appena questi si manifestano**
- **Con CI e le pratiche correlate si riesce a creare una rete di qualità, che ci permette di fornire software di qualità più velocemente**

## Ridurre i rischi (2)

- Consideriamo i seguenti rischi
  - Software non deployable
  - Difetti scoperti tardi
  - Mancanza di visibilità del progetto
  - Software di bassa qualità
- Non si tratta di rendere attenti verso questi rischi (sono tutti noti), ma di permetterne la gestione

## Ridurre i rischi (3)

- Software non “deployabile”, impossibile creare il build

- Riesco a creare il build solo sulla mia macchina

In questo caso è importante sottolineare che il build dev'essere creato in modo indipendente dalla macchina, IDE utilizzato, configurazione specifica, ecc.  
È importante avere un server di integrazione, che usi uno script di build (ant) indipendente.

- Sincronizzazione con il DB

I test devono poter girare utilizzando l'ultima versione dello schema di DB. Lo schema di DB e i suoi dati minimi indispensabili devono trovarsi nel repository.  
I test devono essere in grado di eseguire un “drop” del DB e poi ricrearlo nuovo.

## Ridurre i rischi (4)

- **Difetti scoperti tardi**

- **Regression test**

Sappiamo che dobbiamo avere suite di test nel nostro progetto.  
Basta aggiungere la chiamata di questi test nello script di build, in questo modo verranno eseguiti dal server di integrazione ad ogni modifica.

- **Test coverage**

Esistono tool per verificare la percentuale di copertura dei test.  
Questi tool possono essere associati al processo di CI.

## Ridurre i rischi (5)

- **Mancanza di visibilità**

- **Informazioni**

È necessario che ogni sviluppatore nel progetto sia informato su ogni singola modifica effettuata al progetto.  
Al processo di CI può essere associata una notifica via email sulle modifiche effettuate o sugli eventuali problemi.

- **Visualizzazione grafica della struttura**

Se si desidera avere a disposizione l'ultima versione grafica della struttura del software (UML class diagram), lo si può fare inserendo nel sistema di CI la generazione a partire dall'ultima versione (esempio: Doxygen).

## Ridurre i rischi (6)

- **Software di bassa qualità (1)**

- **Aderenza del codice a standard predefiniti**

L'aderenza agli standard viene spesso controllata manualmente. In realtà esistono tool come Checkstyle, PMD o anche Sonar, che permettono di fare delle verifiche statiche del codice a partire da regole. Questi possono essere integrati al sistema CI.

- **Aderenza all'architettura**

Anche a livello di architettura ci possono essere delle guideline da seguire, ad esempio, il codice del data layer che non accede al codice del business layer, ecc.

Anche queste possono essere controllate da tool (JDepend, NDepend, ecc.) integrabili nel processo di CI.

## Ridurre i rischi (7)

- **Software di bassa qualità (2)**

- **Duplicazione del codice**

Codice duplicato rende più difficili le modifiche e la manutenzione del progetto.

È difficile scoprire dove abbiamo duplicazione di codice all'interno del progetto. Più il progetto è grande, più sviluppatori lavorano e maggiore è la probabilità di avere codice in più parti che esegue la stessa funzionalità.

Anche in questo caso esistono tool (utility CPD del tool di analisi metriche PMD, Simian, ecc.) che possono essere integrati nel processo di CI

## Come introdurre CI (1)

- Tutte le pratiche viste fin qui servono a trarre il massimo beneficio da CI
- Per iniziare non serve però applicarle tutte assieme
- Il primo passo è senza dubbio quello dell'automazione del processo di build
  - Mettiamo il progetto sotto il controllo di un sistema di gestione dei sorgenti (esempio: Subversion)
  - Facciamo in modo che con un unico comando si possa creare un build

## Come introdurre CI (2)

- Il passo successivo potrebbe essere quello di introdurre suite di test nel build automatico
- Se si hanno già test di unità nel sistema, la cosa è molto semplice, basta richiamarli durante il build
- Inizialmente il build verrà fatto partire a mano. Poi, si potrà richiamarlo automaticamente, con uno script, una volta al giorno per il nightly build
- Come ultimo passaggio, possiamo installare un server di integrazione con un software di CI (esempio: TeamCity)

## Introductory Example

### **Interaction between design and code: a starting example**

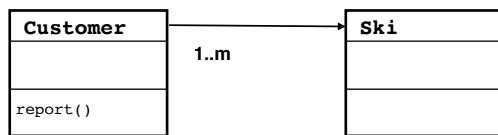
## **Initial Problem**

Program able to compute and prepare a report on the due amount of a customer who is renting different types of skis in a sports' shop.

The program manages, for each customer, which and how many skis have been rented for a certain period of time.

With all these information and different ski types, the program is able to prepare a report, to be used as **invoice**

## Initial structure

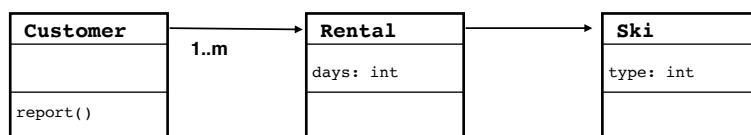


### Problems:

- Management of time (days: in Customer or in Ski?)
- More objects for the same type of ski

## Modified structure

### Intermediate class for the time management



## Ski class

```
public class Ski {
    public static final int MONO = 0;
    public static final int CARVING = 1;
    public static final int CHILDREN = 2;

    private String name;
    private int length;
    private int type;

    public Ski(String name, int length, int type){
        this.name = name;
        this.length = length;
        this.type = type;
    }

    public int getType(){
        return type;
    }

    ...
}
```

## Renting class

```
public class Rental{
    private Ski ski;
    private int days;

    public Rental(Ski ski, int days){
        this.ski = ski;
        this.days = days;
    }

    public int getDays(){
        return days;
    }

    public Ski getSki(){
        return ski;
    }
}
```

## Customer class

```

public class Customer{
    private String name;
    private List<Rental> rentals;

    public Customer(String name){
        this.name = name;
        rentals = new ArrayList<>();
    }

    public String getName(){
        return name;
    }

    public void addRenting(Rental rental){
        rentals.add(rental);
    }

    ...

    public String report(){
        String text = "Rental report for customer " + name+ "\n";
        ...
        return text;
    }
}

```

Sandro Pedrazzini

Starting example 7

## Method *report()*

```

public String report(){
    String text= "Rental report for customer " + name + "\n";
    double totalCost = 0;
    int points = 0;
    for (Rental eachRental : rentals){
        double rentalCost = 0;
        switch(eachRental.getSki().getType()){ //add amount for each rented ski pair
            case Ski.MONO: //25+15 each day after the 2. day
                rentalCost += 25;
                if (eachRental.getDays() > 2)
                    rentalCost += (eachRental.getDays() - 2) * 15;
                break;
            case Ski.CARVING: // 25 each day
                rentalCost += eachRental.getDays() * 25;
                break;
            case Sci.CHILDREN: // 15 * length in meters * #days
                double base = 15 * (double) eachRental.getSki().getLength()/100;
                rentalCost += base;
                if (eachRental.getDays() > 3)
                    rentalCost += (eachRental.getDays() - 3) * base;
                break;
        }
        points++;
        if (eachRental.getSki().getType() == Ski.CHILDREN) points++;
        else if ((eachRental.getSki().getType() == Ski.CARVING) &&
                 (eachRental.getDays() > 1)) //extra 2 bonus carving for >= 2 days
            points += 2;
        text += "\t" + eachRental.getSci().getName()+"\t" + rentalCost + "\n";
        totalCost += rentalCost;
    }
    text += "Due amount: Fr. " + totalCost + "\n";
    text += "You earned " + points + " bonus points\n";
    return text;
}

```

Sandro Pedrazzini

Starting example 8

**Using it**

```
public class Main {
    public static void main(String[] args){
        Rental rental1 = new Rental(new Ski("Stöckli 2B",170,Ski.MONO), 4);
        Rental rental2 = new Rental(new Ski("Völkl BX",110,Ski.CHILDREN), 5);
        Rental rental3 = new Rental(new Ski("K2",180,Ski.CARVING), 2);
        Customer customer = new Customer("John Smith");
        customer.addRental(rental1);
        customer.addRental(rental2);
        customer.addRental(rental3);
        System.out.println(customer.report());
    }
}
```

**This produces following output:**

```
Rental report for customer John Smith
    Stöckli 2B      55.00
    Völkl BX      49.50
    K2            50.00
```

Due amount: Fr. 154.50

You earned 5 bonus points

**Comments**

- This is not a real object-oriented solution, considering design and implementation.
- The method *report()* of class *Customer* has too much responsibility.  
Many operations should be part of other classes.
- It works, why should I modify it?
- How to verify, automatically, that the computed amounts are correct?

**First request:** Add a function able to prepare the report for each customer in HTML

New method *reportHtml()* to add to class *Customer*?  
Redundancies?

**Second request:** New ski classification

**-> Refactoring/restructuring needed, possibly OO**

## Extract Method (1)

If you want a more readable and reusable code, you need to work with short and clear methods

```
public String report(){
    String text = "Rental report for customer" + name + "\n";
    double totalCost = 0;
    int points = 0;
    for (Rental eachRental: rentals){

        double rentalCost = amountOf(eachRental);

        points++;
        if (eachRental.getSki().getType() == Ski.CHILDREN) points++;
        else if ((eachRental.getSki().getType() == Ski.CARVING) &&
                  (eachRental.getDays() > 1))
            points += 2;
        text += "\t" + eachRental.getSki().getName() + "\t" + rentalCost + "\n";
        totalCost += rentalCost;
    }
    text += "Due amount:\t Fr. " + totalCost + "\n";
    text += "You earned " + points + " bonus points\n";

    return text;
}
```

---

Sandro Pedrazzini

Starting example 11

## Extract Method (2)

**New method, extracted from *report()***

```
private double amountOf(Rental eachRental) {
    double cost = 0;
    switch(eachRental.getSki().getType()){
        case Ski.MONO:          // 25 + 15 each day after the second day
            cost += 25;
            if (eachRental.getDays() > 2)
                cost += (eachRental.getDays() - 2) * 15;
            break;
        case Ski.CARVING:        // 25 each day
            cost += eachRental.getDays() * 25;
            break;
        case Ski.CHILDREN:       // 15 * length in meters * # of days
            double base = 15 * (double) eachRental.getSki().getLength()/100;
            cost += base;
            if (eachRental.getDays() > 3)
                cost += (eachRental.getDays() - 3) * base;
            break;
    }

    return cost;
}
```

---

Sandro Pedrazzini

Starting example 12

## Extract Method (3)

**Adapting variable names to new context**

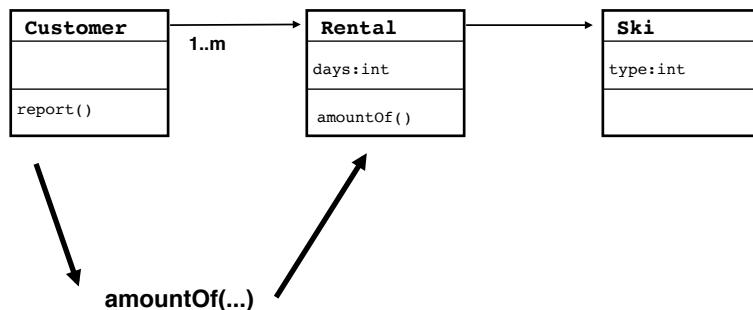
```
public class Customer{
    ...
    private double amountOf(Rental rental) {
        double cost = 0;
        switch(rental.getSki().getType()){
            case Ski.MONO:
                cost += 25;
                if (rental.getDays() > 2)
                    cost += (rental.getDays() - 2) * 15;
                break;
            case Ski.CARVING:
                cost += rental.getDays() * 25;
                break;
            case Ski.CHILDREN:
                double base = 15 * (double) rental.getSki().getLength()/100;
                cost += base;
                if (rental.getDays() > 3)
                    cost += (rental.getDays() - 3) * base;
                break;
        }
        return cost;
    }
}
```

Sandro Pedrazzini

Starting example 13

## Move Method (1)

The method *amountOf()* is part of *Customer*, but it uses only information from *Rental* (and *Ski*).



Sandro Pedrazzini

Starting example 14

## Move Method (2)

In this case, moving the method to Rental, you eliminate the parameter  
The method name should also be readapted to the new context

```
public class Rental {
    ...
    public double computeAmount() {
        double result = 0;
        switch(ski.getType()){
            case Ski.MONO:
                result += 25;
                if (days > 2)
                    result += (days - 2) * 15;
                break;
            case Ski.CARVING:
                result += days * 25;
                break;
            case Ski.CHILDREN:
                double base = 15 * (double)ski.getLength()/100;
                result += base;
                if (days > 3)
                    result += (days - 3) * base;
                break;
        }
        return result;
    }
}
```

## Move Method (3)

Changes within the calling method (*report()*)

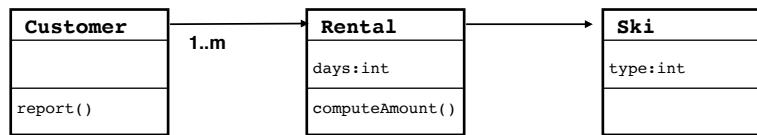
Before:

```
rentalCost = amountOf(eachRental);
```

After:

```
rentalCost = eachRental.computeAmount();
```

## New Structure



Sandro Pedrazzini

Starting example 17

## Checking amounts

- Observing the structure you can remark how it is now easier to check if the computed amount is correct**

```

Rental rental1 =
    new Rental (new Ski("Stöckli 2B", 170, Ski.MONO), 4);
assertEquals(55.0, rental1.computeAmount(), 0.01) //25 + 2*15 => 55

Rental rental2 =
    new Rental (new Ski("Völkl BX", 110, Ski.CHILDREN), 5);
assertEquals(49.5, rental2.computeAmount(), 0.01) //15*1.1*3

Rental rental3 =
    new Rental(new Ski("K2", 180, Ski.CARVING), 2);
assertEquals(50.0, rental3.computeAmount(), 0.01) //25*2
  
```

Sandro Pedrazzini

Starting example 18

## Extract method (1)

The same operation can be performed for the points computation.

```
public String report(){
    String text = "Rental report for customer " + getName() + "\n";
    double totalCost = 0;
    int points = 0;
    for (Rental eachRental: rentals){
        double rentalCost = eachRental.computeAmount();
        puntiAccumulati++;
        if (eachRental.getSki().getType() == Ski.CHILDREN) points++;
        else if ((eachRental.getSki().getType() == Sci.CARVING) &&
            (eachRental.getDays() > 1))
            puntiAccumulati+=2;
        text += "\t" + eachRental.getSki().getName() + "\t" + rentalCost + "\n";
        totalCost += rentalCost;
    }

    text += "Due amount:\t Fr. " + totalCost + "\n";
    text += "You earned " + points + " bonus points\n";

    return text;
}
```

Sandro Pedrazzini

Starting example 19

## Extract method (2)

```
public String report(){
    String text = "Rental report for customer " + getName() + "\n";
    double totalCost = 0;
    int totalPoints = 0;
    for (Rental eachRental: rentals){
        double rentalCost = eachRental.computeAmount();
        int rentalPoints= eachRental.computePoints();
        text += "\t" + eachRental.getSki().getName() + "\t" + rentalCost+ "\n";

        totalCost += rentalCost;
        totalPoints += rentalPoints;
    }

    text += "Due amount:\t Fr. " + totalCost + "\n";
    text += "You earned "+ totalPoints + " bonus points\n";

    return text;
}
```

Sandro Pedrazzini

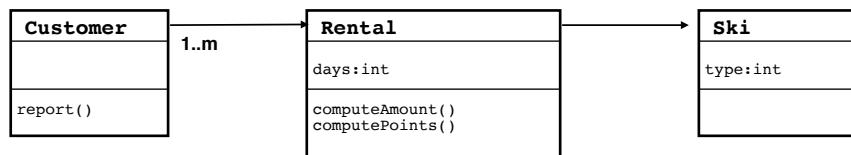
Starting example 20

### Extract method (3)

Also `computePoints()` will be part of `Rental`

```
public class Rental{
    ...
    int computePoints(){
        if (getSki().getType() == Ski.CHILDREN) {
            return 2;
        else if (getSki().getType() == Ski.CARVING) && (days> 1)) {
            return 3;
        } else {
            return 1;
        }
    }
}
```

### New structure



## Checking points

- Also the points computation can be checked separately and in an easier way.

```
Rental rental1 =
    new Rental(new Ski("Stöckli 2B", 170, Ski.MONO), 4);

assertEquals(1, rental1.computePoints());


Rental rental2 =
    new Rental(new Ski("Völkl BX", 110, Ski.CHILDREN), 5);

assertEquals(2, rental2.computePoints());


Rental rental3 =
    new Rental(new Sci("K2", 180, Ski.CARVING), 2);

assertEquals(2, rental3.computePoints());
```

## Replace temporary with query (1)

If we observe the content of the method *report()* we realize that there are two main computations:

- total amounts
- total points

Both computations can be externalized.

The only “problem” is that *report()* also needs the amount of each single rental and so, it uses the same loop needed to compute the total amount.

## Replace temporary with query (2)

The program will be always simplified if we substitute temporary variables with method calls returning the same values.

```

...
double totalCost = 0;
int totalPoints = 0;

for (Rental eachRental: rentals){
    double rentalCost= eachRental.computeAmount();
    int rentalPoints = eachRental.computePoints();
    ...
    totalCost += rentalCost;
    totalPoints += rentalPoints;
}

... totalCost
... totalPoints

```

## Replace temporary with query (3)

- A separate query for total amount and total points:**

- ⇒ Reduces the complexity of the code, avoiding temporary variables
- ⇒ Increases the opportunities of factorization
- ⇒ Favors the test

- In this particular case it is also useful to provide two more methods:**

- `computeTotalAmount()`
- `computeTotalPoints()`

## Replace temporary with query (4)

We eliminate “rentalCost”, “rentalPonts”, “totalCost” and “totalPoints”  
 We add: *computeTotalAmount()* e *computeTotalPoints()*

```
public String report(){
    String text = "Rental report for customer " + getName() + "\n";
    for (Rental eachRental: rentals){
        text += "\t" + eachRental.getSki().getName() + "\t" +
            eachRental.computeAmount() + "\n";
    }

    text += "Due amount:\t Fr. " + computeTotalAmount() + "\n";
    text += "You earned" + computeTotalPoints() + " bonus points\n";

    return text;
}
```

## Replace temporary with query (4)

```
public double computeTotalAmount(){
    double result = 0;
    for (Rental eachRental: rentals){
        result += eachRental.computeAmount();
    }

    return result;
}

public int computeTotalPoints(){
    int result = 0;
    for (Rental eachRental: rentals){
        result += eachRental.computePoints();
    }

    return result;
}
```

## Replace temporary with query (5)

- Better (Java 8 features, in order to reduce boilerplate code)

```
public double computeTotalAmount(){
    return rentals.stream()
        .mapToDouble(r -> r.computeAmount())
        .sum();
}

public int computeTotalPoints(){
    return rentals.stream()
        .mapToInt(r -> r.computePoints())
        .sum();
}
```

## New method

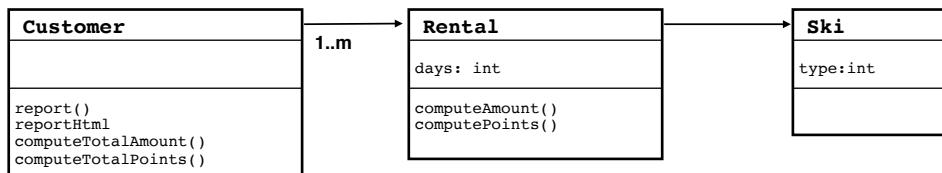
```
public String reportHtml(){
    String text = "<H1>Rental report for " + getName() + "</H1>\n";
    for (Rental eachRental: rentals){
        text += "\t" + eachRental.getSki().getName() + "\t" +
            eachRental.computeAmount() + "<BR>\n";
    }

    text += "<P>Due amount:\t Fr. " +
        computeTotalAmount() + "</P>\n";
    text += "<P>You earned <EM>" + computeTotalPoints() +
        "</EM>bonus points</P>\n";

    return text;
}
```

## New structure

You can remark that the `computeTotalAmount()` and `computeTotalPoints()` methods can be used by whichever other object or method



## Checking total amounts

- We can easily check also the total amounts

```

Rental rental1 =
    new Rental(new Ski("Stöckli 2B", 170, Ski.MONO), 4);
Rental rental2 =
    new Rental(new Ski("Völkl BX", 110, Ski.CHILDREN), 5);
Rental rental3 =
    new Rental(new Ski("K2", 180, Ski.CARVING), 2);

Customer customer = new Customer("John Smith");
customer.addRental(rental1);
customer.addRental(rental2);
customer.addRental(rental3);

assertEquals(104.50, customer.computeTotalAmount(), 0.01);
assertEquals(5, customer.computeTotalPoints());
  
```

## Comments

- We refactored the code so that the implementation of the new method `reportHtml()`, did not create duplications in the “business logic” code
- Some refactorings could have caused performance problems:  
better let discover them by a profiler...

**Next request:** New ski classification

## On code “optimization”

- Donald E. Knuth
  - **We should forget about small efficiencies, say about 97% of the time. Premature optimization is the root of all evil**.
- M.A. Jackson
  - **Rule 1: Do not optimize.**
  - **Rule 2 (only for experts): do not do it yet, at least until you'll reach a good and not optimized solution, perfectly clear.**

## Move the “switch” (1)

SUPSI

```
public class Rental{  
    ...  
    public double computeAmount() {  
        double result = 0;  
        switch(ski.getType()){  
            case Ski.MONO:  
                result += 25;  
                if (days > 2)  
                    result += (days - 2) * 15;  
                break;  
            case Ski.CARVING:  
                result += days * 25;  
                break;  
            case Ski.CHILDREN:  
                double base = 15 * (double)ski.getLength()/100;  
                result += base;  
                if (days > 3)  
                    result += (days - 3) * base;  
                break;  
        }  
        return result;  
    }  
}
```

The main switch statement is located in a class (Rental), different from where the information on the classification type (Ski) is located.

Sandro Pedrazzini

Starting example 35

## Move the “switch” (2)

SUPSI

In order to move the switch to Ski, we pass a new parameter (days), i.e. the unique information we need from Rental

```
public class Ski {  
    ...  
    double computeAmount(int days){  
        ...  
    }  
}
```

We need 2 kinds of information: rental length (from class Rental) and information on Ski type (from class Ski).

Because we want to refactor the code in order to be able to modify the latter in the easiest way, we'd better move the method to Ski.

Sandro Pedrazzini

Starting example 36

### Move the “switch” (3)

```
public class Ski {
    ...
    public double computeAmount(int days) {
        double result = 0;
        switch(type){
            case Ski.MONO:
                result += 25;
                if (days > 2)
                    result += (days - 2) * 15;
                break;
            case Ski.CARVING:
                result += days * 25;
                break;
            case Ski.CHILDREN:
                double base = 15 * (double)getLength()/100;
                result += base;
                if (days > 3)
                    result+= (days- 3) * base;
                break;
        }
        return result;
    }
}
```

Sandro Pedrazzini

Starting example 37

### Move the “switch” (4)

- Moving the method to *Ski* does not have to mean eliminating it in *Rental*.
- On the opposite: it is a good thing if *Rental* still offers this functionality in its interface, delegating to *Ski* the answer

```
public class Rental {
    ...
    public double computeAmount(){
        return ski.computeAmount(days);
    }
}
```

Sandro Pedrazzini

Starting example 38

## Move the “switch” (5)

Also the method `Rental.computePoints()` uses information on `Ski` types for its decision, it is then also involved in case of changes in classifications. Thus: better to move it to `Ski` too.

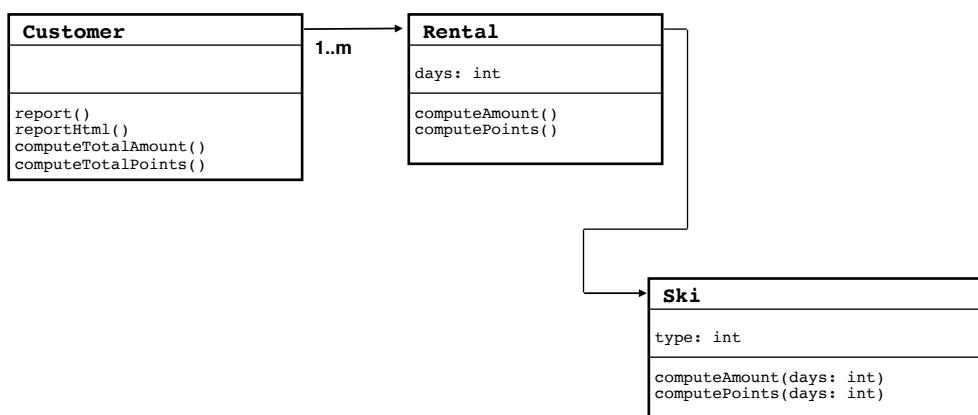
```
public class Rental{
    ...
    public int computePoints(){
        return ski.computePoints(days);
    }
}

public class Ski {
    ...
    public int computePoints(int days){
        if (type == Ski.CHILDREN) {
            return 2;
        } else if ((type == Ski.CARVING) && (days > 1)) {
            return 3;
        } else {
            return 1;
        }
    }
}
```

Sandro Pedrazzini

Starting example 39

## New structure



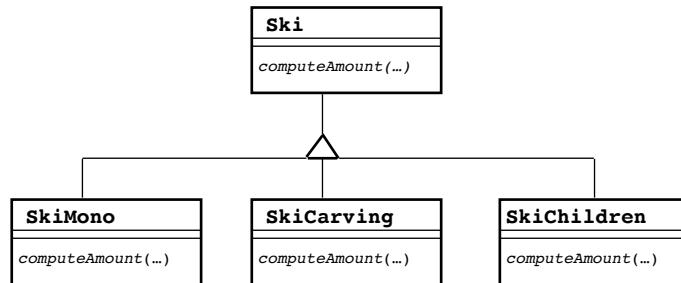
Sandro Pedrazzini

Starting example 40

## Replace conditional logic (1)

We can observe that we have different types of Ski, each of them having a different way of answering to the same question.

Change: 3 Ski subclasses, where each of them implements its own `computeAmount()` method (pattern *Strategy*)

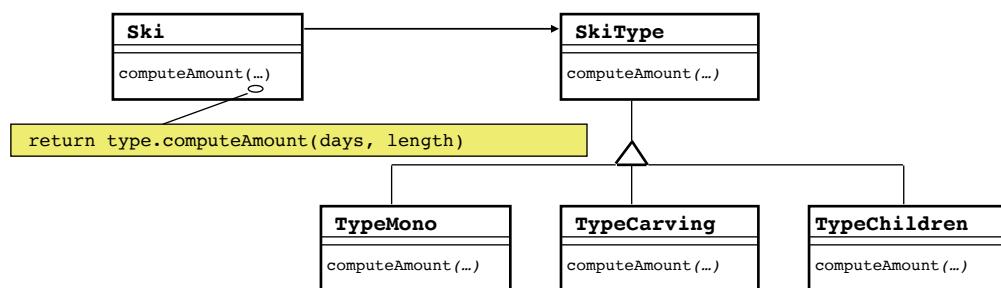


Sandro Pedrazzini

Starting example 41

## Replace conditional logic (2)

Another possible variant:  
delegate the classification logic to a different hierarchy (pattern *State*).



Sandro Pedrazzini

Starting example 42

### Replace conditional logic (3)

#### Code structure within Ski class (Strategy pattern)

```
public abstract class Ski {
    ...
    private String name;
    private int length;

    public Ski(String name, int length){
        this.name = name;
        this.length = length;
    }

    public abstract double computeAmount(int days);
    public abstract double computePoints(int days);
}
```

### Replace conditional logic (4)

In the next step we extract each single “case” and we insert its code into the corresponding subclass

```
public class Skimono extends Ski {
    public Skimono(String name, int length) {
        super(name, length);
    }

    public double computeAmount(int days) {
        double result = 25;
        if (days > 2) {
            result += (days - 2) * 15;
        }
        return result;
    }
    ...
}
```

## Replace conditional logic (5)

```
public class SkiChildren extends Ski {  
    public SkiChildren(String name, int length) {  
        super(name, length);  
    }  
  
    public double computeAmount(int days) {  
        double base = 15 * (double)getLength()/100;  
        double result = base;  
        if (days > 3) {  
            result += (days - 3) * base;  
        }  
        return result;  
    }  
    ...  
}
```

## Replace conditional logic (6)

```
public class SkiCarving extends Ski {  
    public SkiCarving(String name, int length) {  
        super(name, length);  
    }  
  
    public double computeAmount(int days) {  
        return days * 25;  
    }  
  
    ...  
}
```

## Replace conditional logic (7)

The same procedure can be carried on for `computePoints()`, which is also included in the `Ski` hierarchy.

```
public abstract class Ski {
    ...
    public abstract double computeAmount(int days);
    public abstract double computePoints(int days);
}
```

## Replace conditional logic (8)

Overriding of `computePoints()` in the hierarchy

```
public class SkiCarving extends Ski {
    ...
    public double computePoints(int days){
        if (days > 1) {
            return 2;
        } else {
            return 1;
        }
    }
}
```

## Replace conditional logic (9)

```
public class SkiMono extends Ski {
    ...
    public double computePoints(int days){
        return 1;
    }
}

public class SkiChildren extends Ski {
    ...
    public double computePoints(int days){
        return 2;
    }
}
```

## Replace conditional logic (10)

Depending on the different situations, you could still encounter a switch during the object creation phase (let's say we need to create a **SkiChildren** object within the **Rental**'s constructor)

**This is however a limited problem, because it would be a unique switch statement, alone and isolated, without consequences.**

There are, however, some solutions:

- Reflection (pass the class name, instead of a flag)
- Use of Dependency Injection (in our case: external initialization)

## Replace conditional logic (11)

### Use

```
public class Main {
    public static void main(String[] args){
        Rental rental1 =
            new Rental(new SkiMono("Stöckli 2B", 170), 4);
        Rental rental2 =
            new Rental(new SkiChildren("Völkl BX", 110) 5);

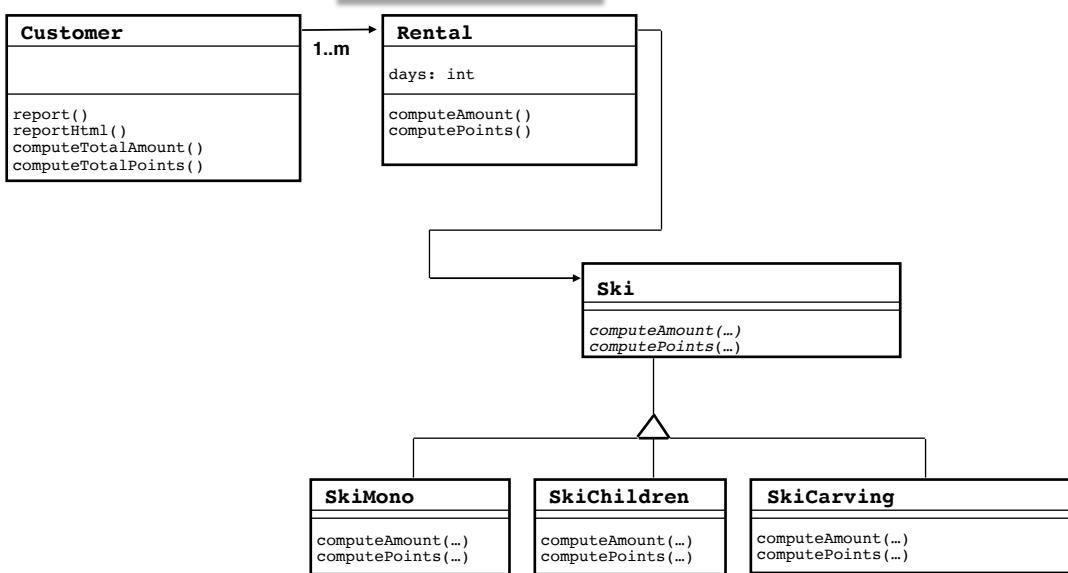
        Customer customer = new Customer("John Smith");
        customer.addRental(rental1);
        customer.addRental(rental2);

        System.out.println(customer.report());
        System.out.println(customer.reportHtml());
    }
}
```

Sandro Pedrazzini

Starting example 51

## New structure



Sandro Pedrazzini

Starting example 52

## Comments

- We eliminated the repeated switch used to determine cost and bonus
- Each change in type, price rules and bonus can now be performed locally, only once, without consequences in the rest of the code.

### Improvements

- Use of pattern *Template* in order to still improve the code of *Customer.report()*

## Situation

We have two methods, performing similar operations, in the same order:

```
report()
reportHtml()
```

### Improvement:

Use of hierarchy in order to eliminate duplications in the behaviour.  
In this case the duplication consists in the operations' sequence

-> *Template* pattern

## Form template method (1)

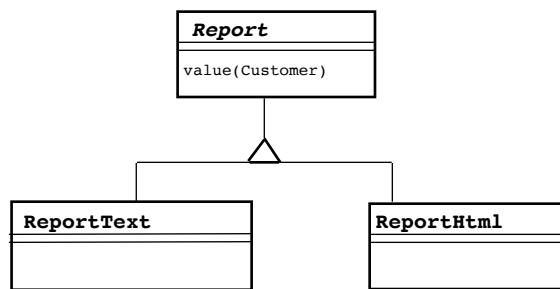
We need first to identify the elements, different, but used in the same sequence:

```
public class Customer {
    ...
    public String report(){
        String testo = "Rental report for " +getName() + "\n";           → header
        for (Rental eachRental: rentals){
            text += "\t" +eachRental.getSki().getName() + "\t" +
                    eachRental.computeAmount() + "\n";                         → rental details
        }
        text += "Due amount:\t Fr. " +computeTotalAmount() + "\n";
        text += "You earned " +computeTotalPoints() +
               "bonus points\n";                                         → footer
        return text;
    }
}
```

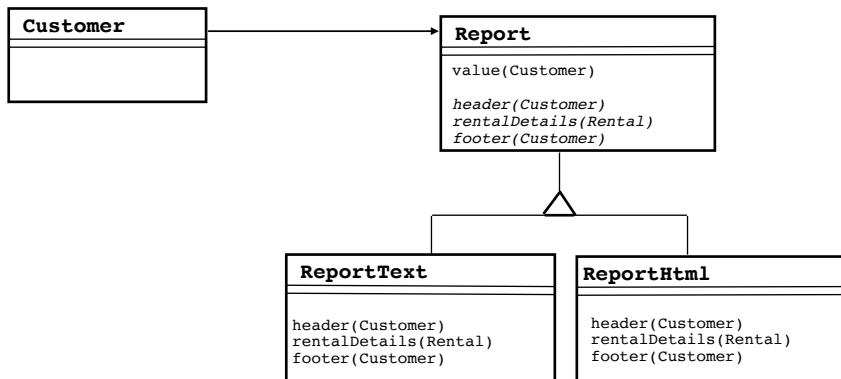
## Form template method (2)

The different elements must become part of two different classes, which have, however, a common superclass.

The superclass contains the common main method that determines the main flow and calls the subclasses' methods for the specific parts (hooks).



## Structure



## Form template method (3)

The next step consists on extracting the different parts from the main methods and create separated methods for the subclasses:

**header(), rentalDetails(), footer()**

The first is the header():

```

public class ReportText extends Report {
    ...

    protected String header(Customer customer){
        return "Rental report for " + customer.getName() + "\n";
    }

    ...
}
  
```

## Form template method (4)

header() for the class *ReportHtml*:

```
public class ReportHtml extends Report{
    ...
    protected String header(Customer customer){
        return "<H1>Rental report for " + customer.getName() + "</H1>\n";
    }
    ...
}
```

## Form template method (5)

**Other element to extract:** details of each single rental

```
protected String rentalDetails(Rental rental){
    return "\t" + rental.getSki().getName() + "\t" +
           rental.computeAmount() + "\n";
}

protected String rentalDetails (Rental rental){
    return "\t" + rental.getSki().getName() + "\t" +
           rental.computeAmount () + "<BR>\n";
}
```

## Form template method (6)

**Other element to extract:** footer

```
protected String footer(Customer customer) {
    return "Due amount:\t Fr. "
        + customer.computeTotalAmount() +
        "\n" + "You earned "
        + customer.computeTotalPoints() +
        " points\n";
}

protected String footer(Customer customer) {
    return "<P>Due amount:\t Fr. "
        + customer.computeTotalAmount() +
        "</P>\n" +
        "<P>You earned <EM>"
        + customer.computeTotalPoints() +
        "</EM> points</P>\n";
}
```

## Form template method (7)

**The common element (template) can be specified once in the *Report* superclass**

```
public abstract class Report {

    public String value(Customer customer){
        String text = header(customer);
        for (Rental eachRental: customer.getRentals()){
            text += rentalDetails(eachRental);
        }
        text += footer(customer);

        return text;
    }

    abstract protected String header(Customer customer);
    abstract protected String rentalDetails(Rental rental);
    abstract protected String footer(Customer customer);
}
```

## Integration

SUPSI

```
public class Customer{  
    ...  
    public String report() {  
        return new ReportText().value(this);  
    }  
    public String reportHtml() {  
        return new ReportHtml().value(this);  
    }  
    ...  
}
```

Sandro Pedrazzini

Starting example 63

## Main

SUPSI

```
public class Main {  
  
    public static void main(String[] args){  
        Rental rental1 =  
            new Rental(new SkiMono("Stöckli 2B", 170),4);  
        Rental rental2 =  
            new Rental(new SkiChildren("Völkl BX", 110),5);  
  
        Customer customer = new Customer("John Smith");  
        customer.addRental(rental1);  
        customer.addRental(rental2);  
  
        System.out.println(customer.report());  
        System.out.println(customer.reportHtml());  
    }  
}
```

Sandro Pedrazzini

Starting example 64

# Software Engineering: Design Patterns I

## 1. Iterator

Write a class *MyList* representing a generic list, reusing the available Java *ArrayList* through composition having following public methods:

```
MyList<T>();
void addElement(T ob);
int length();
T getElement(int pos);
MyIterator<T> getForwardIterator();
MyIterator<T> getBackwardIterator();
```

Create for this list two iterators.

*ForwardIterator* must allow the iteration on the list from the first to the last element, whereas *BackwardIterator*, must allow the iteration from the last to the first element.

Both iterators must be implemented using following *interface*:

```
public interface MyIterator<T> {
    void rewind();
    T nextElement();
    boolean hasMoreElements();
}
```

Write all needed JUnit tests for the iterators and for the list.

## 2. Use of iterators

Realize a small program able to read, line by line, the file content of a text file (which filename is simply specified as hard-coded name within the code, or could be passed through the command line), printing the lines in backward order to the standard output.

Verify that the program works correctly using your version of *MyList* and iterators.

# **Software Engineering: Design Patterns II**

## **1. Changes**

Evaluate how to integrate into the code realized in the preceding lab, exercise 2, following requests of changes:

1. Modify the code, so that the output will be shown using upercases.
2. Modify the code, so that the output will be written, in the same reading order (forward), into a further text file.
3. Modify the code, so that after the output of the file content, the program writes to standard output how many 'a' letters are contained in the text.

## **2. Generalizations**

In order to better evaluate how to integrate them into the code, try to think of following generalizations, applied to the previous changes:

1. The output can be converted into uppercase, lowercase, or other (also future) conversion combinations. The choice of the conversion can be specified hardcoded into the main program, but adding new conversion types does not require changes in the code used to print the output.
2. You want to give the choice to decide whether to print the output to a file, to standard output, or both (ev. to an unlimited number of files), and in which line order (forward or backward).
3. You want to give the choice whether to show or not statistics (more than one, and which ones) at the end of the output.

The changes, with their generalizations, should lead to a code restructuring, adding more classes and abstractions.

## Design Patterns I

### Introduction to Design Patterns

## Motivation

To use and exploit the object-oriented programming is not easy.

A flexible and reusable design is not easy to obtain at the first time, you need more iterations to get it.

We need the opportunity to exploit design reuse, not only code

### General arguments:

- The problems are always very specific and you need specific software to solve them
- If you want to reuse software, you need to write more generic code
- **You do not need to obtain the most generic result the first time, you can start with a specific solution and move it to a more generic one during further iterations**

## Design Patterns

### First definition

The design patterns describe simple and elegant solutions to the specific problems of object-oriented design

The use of patterns requires a certain amount of time more than the "ad hoc" solutions, but that more work is what increases the degree of reuse and flexibility of the software.

## OO Programming

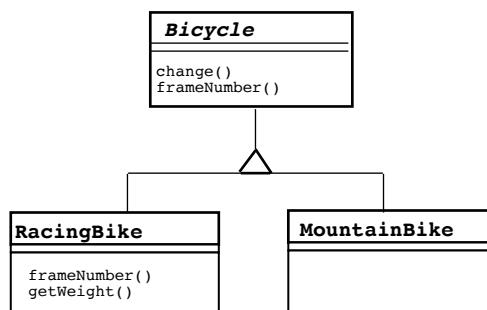
### Some key characteristics

- Concepts of class and object
- Information hiding
- Inheritance
- Polymorphism
- Dynamic binding
- ...

## Inheritance (1)

```
public class RacingBike extends Bicycle {  
    private float weight;  
  
    public RacingBike(int gears, float weight){  
        super(gears);  
        this.weight = weight;  
    }  
  
    public String frameNumber(){  
        return "000";  
    }  
  
    public float getWeight(){  
        return weight;  
    }  
}
```

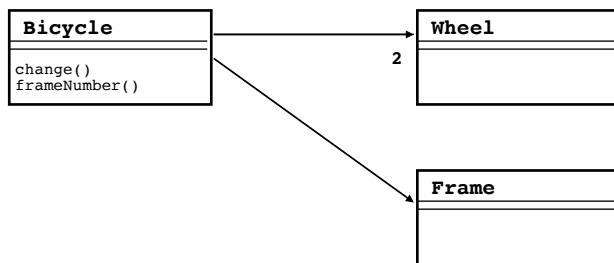
## Inheritance (2)



## Aggregation / composition (1)

```
public class Bicycle {  
    private Frame frame;  
    private Wheel frontWheel, backWheel;  
    private int gears;  
  
    public Bicycle(int gears){  
        frame = new CarbonFrame();  
        frontWheel = new LenticularWheel();  
        backWheel = new LenticularWheel ();  
        this.gears = gears;  
    }  
  
    public int getGears(){  
        return gears;  
    }  
  
    public String frameNumber(){  
        return frame.getNumber();  
    }  
    ...  
}
```

## Aggregation / composition (2)



**Introduction** **SUPSI**

## Aggregation / composition (3)

```

graph LR
    Frame[Frame] --> Pane[Pane]
    Pane --> Tab[Tab]
    Tab --- Button[Button]
    Tab --- List[List]
    Tab --- Popup[Popup]
    Tab --- Label[Label]
    Tab --- Ellipsis[...]
  
```

Appointment Scheduling

Contract Number	Insurance Policy	Customer First...	Customer Last ...	Vehicle Make	Vehicle Model	Commitmable
0000000625	182/001	Leanne	Thurman	VAUXHALL	CAVALIER	<input type="checkbox"/>
0000000626	158/001	Leanne	James	VAUXHALL	TIGRA SPORT	<input type="checkbox"/>
0000000627	177/001	Robert	Wing	OKA	330 I SE AUTO	<input type="checkbox"/>
0000000628	206/001	Lea	Thurman	RENAULT	MEGANE ALIZE	<input type="checkbox"/>
0000000630	227/001	Clark	Kern	NISSAN	PRIMERA SI	<input type="checkbox"/>
0000000631	218/001	Ben	Rogers	VAUXHALL	CORSA .COM 1	<input type="checkbox"/>
0000000633	256/001	Mike	White	VAUXHALL	MAGNUM	<input type="checkbox"/>
0000000634	385/001	Harry	Days	SKODA	OCTAVIA ELEG	<input type="checkbox"/>
0000000635	545/001	Kevin	McGinty	Ford	Focus	<input type="checkbox"/>
0000000641	606/1228945...	John	Smith	Opel	Corsa	<input type="checkbox"/>
0000000642	878/1228945...	Peter	Black	Opel	Corsa	<input type="checkbox"/>
0000000683	1227422654...	John	Smith	Renault	Clio	<input type="checkbox"/>

Device Installation Appointment

Date (none)  
Address  
Dealer CAR TECH SOUND AND SECURITY

---

Sandro Pedrazzini

Design patterns

9

**Introduction** **SUPSI**

## Polymorphism

```

classDiagram
    class Bicycle {
        change()
        frameNumber()
    }
    class RacingBike {
        frameNumber()
        getWeight()
    }
    class MountainBike
    Bicycle <|-- RacingBike
    Bicycle <|-- MountainBike
  
```

```

Bicycle bike = new RacingBike();
Frame frame = new CarbonFrame();

void checkBicycle(Bicycle bike){
    ...
    ... = bike.frameNumber();
    ...
}
  
```

---

Sandro Pedrazzini

Design patterns

10

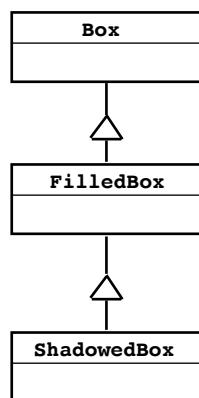
## Polymorphism (2)

### Liskov Substitution Principle (1988)

Practice consequence:

A reference to the base class must be able to use objects of the derived classes without knowing them

## Polymorphism (3)



```
Box b1 = new ShadowedBox();
FilledBox b2 = new ShadowedBox();

void check(Box b){
    ...
}

...
check(b1);
...
check(b2);
```

## Polymorphism and dynamic binding

- Wherever there is a variable declared of the base class, it can be used to reference an object of a derived class (Liskov)
- The implementation of a system must focus on some basic classes
- The knowledge of some specific derived classes must be isolated and limited in fewest points in the code

## Type check

- The dynamic binding can have different kinds of type check:
  - Static check
  - Dynamic check
- **Static check** (Java): it helps to avoid runtime errors, because it ensures, at compile time, that a certain object is able to respond to a message  
=> Java: dynamic binding with static check
- **Dynamic check** (CLOS, Smalltalk): the assignment of an object to a variable is free, there is no type checking at compile time. At runtime, the object must be able to respond to the received messages  
=> “true dynamic binding”

## Aspects that facilitate code reuse



### Abstraction

The abstract solutions are more easily adaptable

### Adaptability

Ability to create new solutions without intervening in existing code

### Centralization of the concrete choice

Decisive for good maintenance

=> Spring, IoC, Dependency Injection

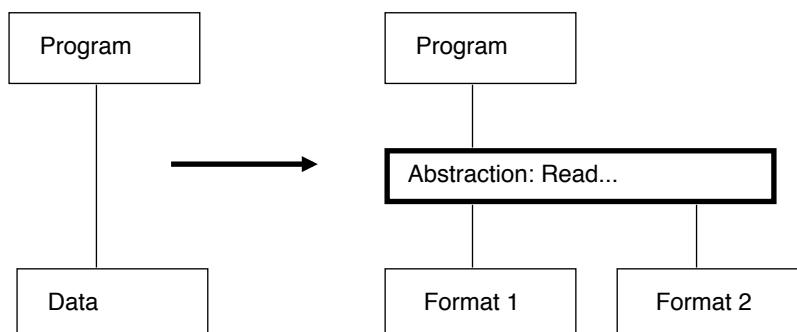
### Standardization

Standard solutions increase the reuse of software

### Trust

Lack of trust leads to the implementation of own solutions ...

## Software extension



## Design experience

- Capitalize on the design experience
- How to reuse the solutions adopted in the past?
- How to store them?

→ Catalogue of the models (patterns) used in the past.  
Each model is reduced to the essential

→ **Design Patterns**

## Use of design patterns

*"Once you understand the design patterns and have had an "Aha!" experience with them, you won't ever think about object-oriented design in the same way."*

*(Gamma et al.)*

### Consequences:

- Programming nearest to the design
- Software documentation at a higher level
- Increased software flexibility
- The concept of reuse becomes effective
- Use the experience of "experts"

## Definitions (1)

- Each pattern (model) gives a name, explains and evaluates a type of recurrent design in OO programming.
- The patterns facilitate and promote the reuse of design and software architectures.
- Documenting new architectures through the patterns makes them more accessible.

## Definitions (2)

- The patterns facilitate any alternative choices in software design.
- The design patterns facilitate the software maintenance
- The patterns bring design closer to programming.  
With patterns, design becomes "abstract programming."

## Elements of a pattern

1. Intent
2. Alternative name
3. Motivation
4. Applicability
  
5. Structure
6. Participants
7. Collaborations
8. Consequences
  
9. Implementation's techniques
10. Sample code
11. Known uses
12. Related patterns

## Example: Proxy (1)

**Intent:** Provide a surrogate or placeholder for another object to control access to it

**Alt. name:** Surrogate

**Motivation:** One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.  
(...)

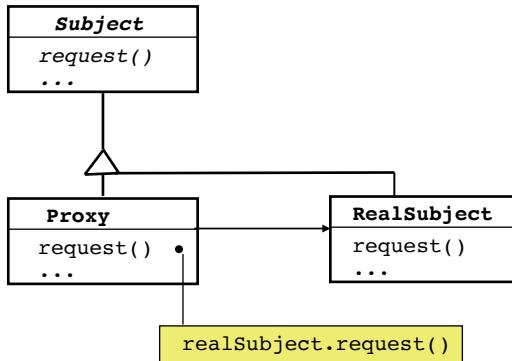
**Applicability:** Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.

**Examples:**

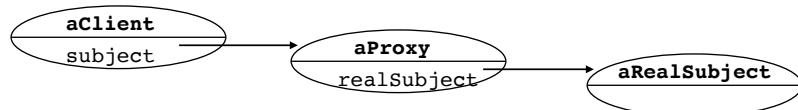
- **Virtual proxy** (creates expensive objects on demand)
- **Remote proxy** (local representative for an object in a different address space)
- **Protection proxy** (controls access to the original object)
- **Smart reference** (replacement for a bare pointer that performs additional actions when an object is accessed)

## Example: Proxy (2)

**Structure:**



Possible proxy structure at runtime



## Example: Proxy (3)

**Participants:**      Proxy

...  
Subject

...  
RealSubject

...

**Collaborations:**      Depending on the kind proxy type, the *Proxy* object forwards requests to the *RealSubject*, when appropriate

**Consequences:**      The *Proxy* pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy  
(...)

**Implementation:**      ...

## Example: Proxy (4)

**Sample code:** ...

**Known uses:** Virtual proxy: ET++ text building block classes

NEXTSTEP uses proxies (instances of class NXProxy) as local representatives for objects that may be distributed.

(...)

**Related patterns:** Adapter (...)  
Decorator (...)

## Sample code (1)

- Program that has to manage information from different text files.
- The information is:
  - Size (in bytes) of the content
  - File name
  - File content, with possible access line by line

## Sample code (2)

- To implement the real functionality we produce a class that deals with the file load and carries out the operations described by the following interface

```
public interface FileInfo {
    public String getFileName();
    public int getSize();
    public String getContent();
    public String getLine(int lineNr);
}
```

## Sample code (3)

- The base functionalities are realized within the class *RealFileInfo*

```
public class RealFileInfo implements FileInfo {
    private String fileName;
    private String content;
    private int byteContentSize;

    public RealFileInfo(String fileName){
        this.fileName = fileName;
        try{
            FileInputStream file = new FileInputStream(fileName);
            byteContentSize = file.available();
            byte[] byteContent = new byte[byteContentSize];
            file.read(byteContent);
            file.close();
            content = new String(byteContent);
        } catch(Exception e){
            ...
        }
    }
    ...
}
```

## Sample code (4)

### RealFileInfo

```

    ...
    public String getFileName() {
        return fileName;
    }

    public String getContent() {
        return content;
    }

    public int getSize() {
        return byteContentSize;
    }
    ...

```

## Sample code (5)

### RealFileInfo

```

    ...

    public String getLine(int lineNr){
        BufferedReader reader = new BufferedReader(new StringReader(content));
        String line = null;
        int currentLineNr = 1;
        try {
            if (lineNr > 0){
                while (((line = reader.readLine())!= null) &&
                       (currentLineNr != lineNr)){
                    currentLineNr++;
                }
            }
        } catch (IOException e) {
            ...
        }

        return line;
    }

```

## Sample code (6)

- Example of use of **RealFileInfo**

```
public class FileInfoTry {  
    public static void main(String[] args){  
        FileInfo fileInfo = new RealFileInfo(args[0]);  
        System.out.println("size: " + fileInfo.getSize());  
        System.out.println("name: " + fileInfo.getFileName());  
        System.out.println("line: 4 " + fileInfo.getLine(4));  
        System.out.println("line: 4 " + fileInfo.getLine(4));  
        System.out.println("line: 6 " + fileInfo.getLine(6));  
    }  
}
```

## Sample code (7)

- The class **RealFileInfo** provides all required functionalities
- Its use, however, is inefficient, because if you would like to have access only to the file name you should not need to load the whole file content into memory
- Another case of inefficiency is when you desire to access multiple times to the same line number: this would be sought and read every time

## Sample code (8)

- The *Proxy* pattern allows you to improve the situation, building up new functionalities, maintaining at the same time the current implementation of *RealFileInfo*, which is needed for the core functionality
- The pattern suggests the implementation of a new class (*ProxyFileInfo*):
  - having the same interface of the original class containing the core functionalities
  - being able to solve the new requests, without wasting resources: only if a more complex request arrives, the proxy would create an instance of the real object (*RealFileInfo*)

## Sample code (9)

- First implementation

```

public class ProxyFileInfo implements FileInfo {
    private FileInfo real = null;
    private String fileName;

    public ProxyFileInfo(String fileName) {
        this.fileName = fileName;
    }

    private FileInfo getReal(){
        if (real == null){
            real = new RealFileInfo(fileName);
        }
        return real;
    }
    ...
}
  
```

## Sample code (10)

```
...
public String getFileName(){
    return fileName;
}

public String getContent(){
    return getReal().getContent();
}

public int getSize() {
    return getReal().getSize();
}

public String getLine(int lineNr){
    return getReal().getLine(lineNr);
}
```

## Sample code (11)

- The file name can be returned without loading the whole file content
- The access to the RealFileInfo is centralized in *getReal()* (the eventual creation is called only once)
- The use of such a proxy can suggest further improvements:
  - Example: cache to manage the direct access to the single lines of text

## Sample code (12)

```

public class ProxyFileInfo implements FileInfo {
    private FileInfo real = null;
    private String fileName;
    private Map<Integer, String> cache = new HashMap<>();

    public ProxyFileInfo(String fileName) {
        this.fileName = fileName;
    }
    ...

    public String getLine(int lineNr){
        String line = cache.get(lineNr);
        if (line == null){
            line = getReal().getLine(lineNr);
            cache.put(lineNr, line);
        }
        return line;
    }
}

```

## Sample code (13)

- The main program, using the implemented functionalities is similar to the one used before for *RealFileInfo*, the only difference is the instantiated object:

```

public class FileInfoTry {
    public static void main(String[] args){
        FileInfo fileInfo = new ProxyFileInfo(args[0]);
        System.out.println("size: " + fileInfo.getSize());
        System.out.println("name: " + fileInfo.getFileName());
        System.out.println("line: 4 " + fileInfo.getLine(4));
        System.out.println("line: 4 " + fileInfo.getLine(4));
        System.out.println("line: 6 " + fileInfo.getLine(6));
    }
}

```

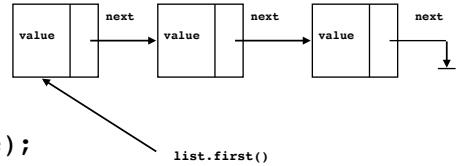
## From problem to pattern: iterator

**1. Array:**

```
for(int i = 0; i < list.length; i++) {
    System.out.println(list[i]);
}
```

**2. Simple dynamic list:**

```
Node node = list.first();
while(node!= null){
    System.out.println(node.value);
    node = node.next;
}
```



### Disadvantages:

- Full exposure of the list implementation
- Full dependency in case of changes in the list's implementation
- Too many dependencies in case of change in the iteration type

## First changes

### Purpose

Make the object client fully independent from the list implementation

### Realization

1. Internal management of the “current” element.
2. List interface extension, in order to manage the iteration internally.

```
void rewind();
T nextElement();
boolean hasMoreElements();
```

### Use

```
list.rewind();
while(list.hasMoreElements()) {
    System.out.println(list.nextElement());
}
```

## Evaluation

### Advantages

- The separation between use and implementation has been obtained
- There is now an abstraction that allows changes in the implementation, and also changes in the semantics

Ex: iterate in reverse order

### Disadvantages

- The management of the “current” item within the list itself is a limit

Ex: Double iteration

## Further changes

### Purpose

Separate the management of the “current” element extracting it from the list.

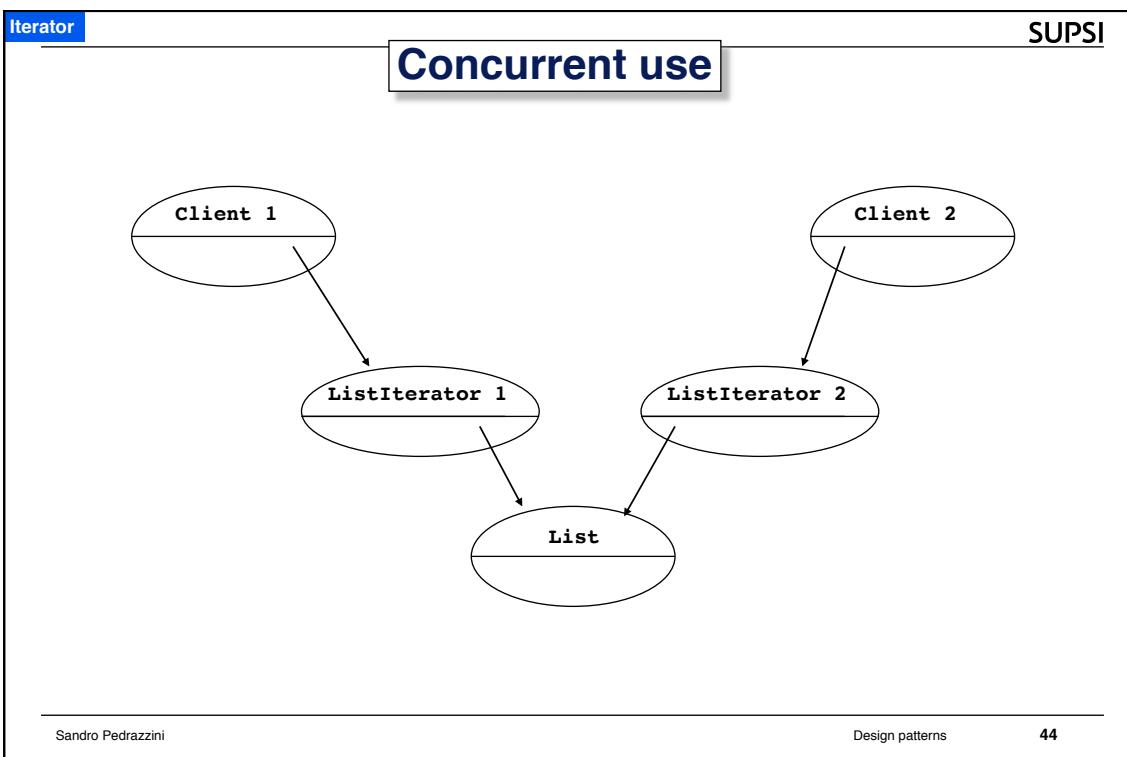
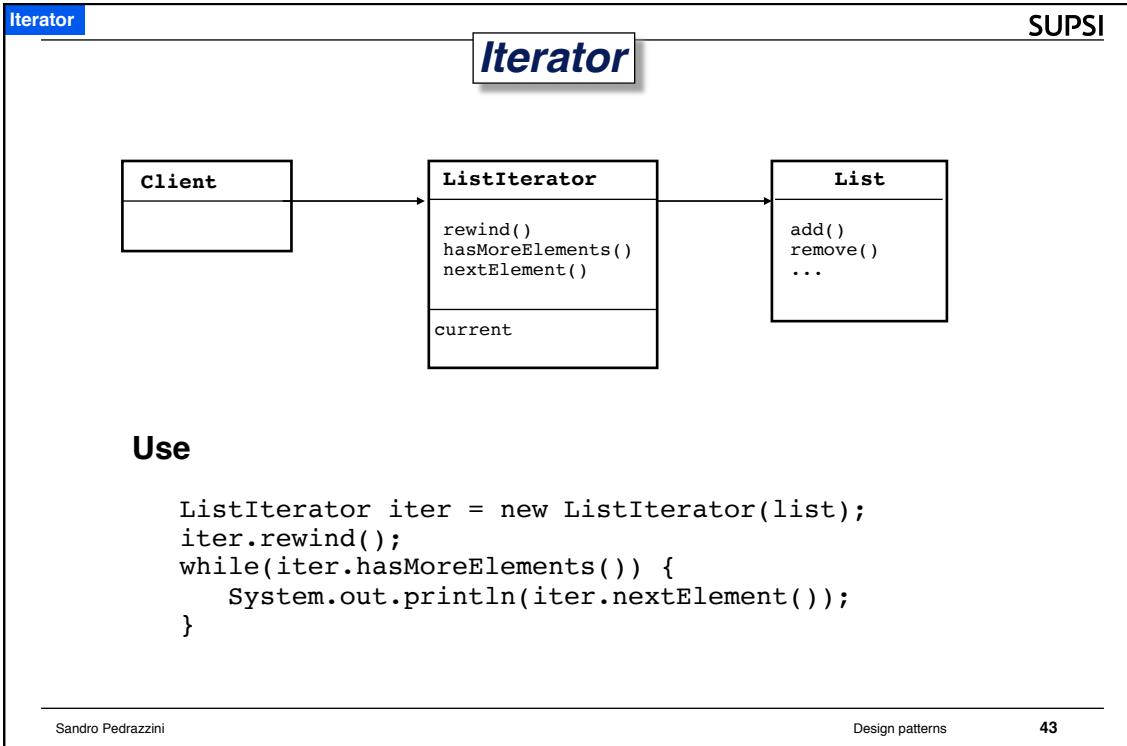
Make it possible to use a single list through more concurrent clients, each of them having its own current element.

Maintain the separation between client and list, as realized in the first step

### Realization

Create an intermediate class between the client and the list to manage the “current” element. This intermediate class, not the list, will now implement the previously specified interface:

```
void rewind();
T nextElement();
boolean hasMoreElements();
```



## Polymorphic use (1)

### Purpose

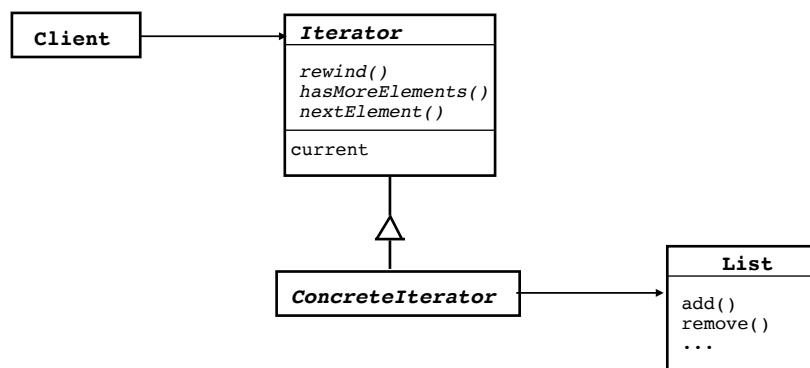
Generalize the concept of iterator allowing the interchange of different iterators within the same code.

### Realization

Create an iterators' hierarchy, in which a common interface is specified for all iterators.

## Polymorphic use (2)

### Structure



### Use

```
Iterator iter = new ConcreteIterator(list);
iter.rewind();
while(iter.hasMoreElements()) {
    System.out.println(iter.nextElement());
}
```

## Iterator

### Consequences of the **Iterator** pattern

It supports changes in the aggregate element to be iterated (not necessarily a list).

-> Just replace the iterator instance with a different one

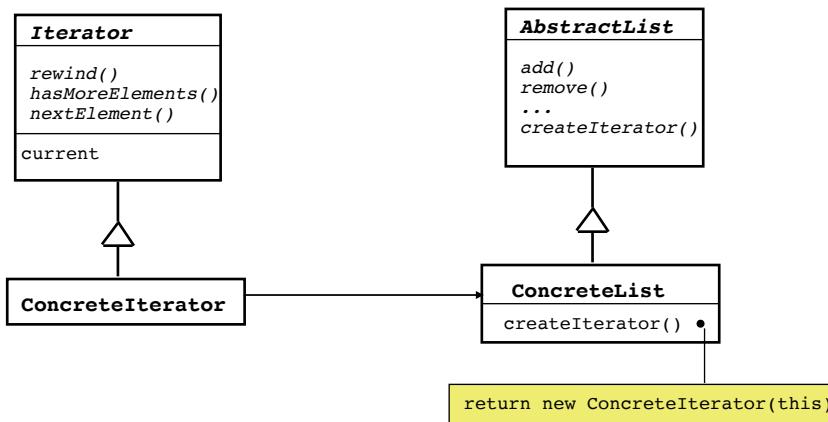
It simplifies the interface of the element to which it refers.

-> It implements internally the access to the element itself

More simultaneous iterations are possible

## Extension (1)

Polymorphic use, not only of the iterator, but also of the aggregated element



## Extension (2)

If each list type has its own specific iterator, the creation of the iterator object can be delegated to the list itself

(-> *Factory Method pattern*)

### Use

```
Iterator iter = list.createIterator();
iter.rewind();
while(iter.hasMoreElements()) {
    System.out.println(iter.nextElement());
}
```

## Principles of OO design

### How to afford the design: divide the code into objects

The composition of a system through objects is one of the difficult parts of object-oriented programming.

Various approaches:

- Focus collaborations and responsibilities within the system, then divide objects into the various elements
- Model the real world so you can translate into objects items arising during the analysis
- Terminology used within the requirements
- Responsibilities and collaborations
- Sequence diagrams
- Design by refactoring
- Design by testing

## Design problems

The rigorous modeling of the real world leads to a system that reflects the current reality, but not necessarily the future one.

The abstractions that are created during the design phases are fundamental for the realization of a flexible system.

The design patterns help in identifying the necessary abstractions and defining the classes that implement them.

## Classes and types (1)

### Class

The class defines how an object is implemented.  
The internal state (variables) and the implementation of the methods..

### Type

The type defines the interface of an object, which requests it is able to respond to.

=> An object can have more than one type.

=> Objects of different classes may belong to the same type.

## Classes and types (2)

```

classDiagram
    class Shape
    class PersistentObj
    class ElasticObject
    class Triangle

    Shape <|-- Triangle
    PersistentObj <|-- Triangle
    ElasticObject <|-- Triangle

```

```

interface PersistentObject {
    void write();
    void read();
}

interface ElasticObject {
    void enlarge(int offset);
    void restrict(int offset);
}

public class Triangle extends Shape
    implements PersistentObject, ElasticObject {
    ...
}

```

## Abstract classes and *interfaces*

- Object manipulation through abstract classes or interfaces**
- The client (the object that "uses") do not know (and should not know) the specific type of the object it uses.
  - The client does not declare variables of a specific class, but variables of abstract classes (or interfaces).
  - Generic programming.  
It reduces implementation dependency between subsystems

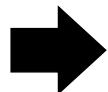
```

Ex:   Iterator iter = new ConcreteIterator(list);
        ...

```

## First principle

The first principle of reusable OO design:



*"Program to an interface,  
not to an implementation"*

## Inheritance and composition

Two different techniques for reusing implementation

### Inheritance:

- *white-box reuse*
- the internal parts of the superclass must be, at least partially, known

### Composition:

- *black-box reuse*
- no internal detail of the used object must be visible

## Comparison (1)

### Inheritance

Static definition (compile-time)

Directly supported by the language

Changes in behavior are simple (overriding)

Impossible, at runtime, to change the hierarchy

With the white-box reuse we break the principles of encapsulation and information hiding

Changes in the superclass can mean adaptations in the subclass.

-> Dependency that limits the reuse

-> A method to avoid this: only inherit from the abstract classes

## Comparison (2)

### Composition

Since the object is accessible only through its interface, you do not break the encapsulation

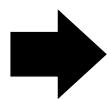
Each object in the composition can be replaced at run-time, if the new object has the same interface (both belongs to the same type)

-> Dynamicity

Less dependence on implementation, if the reference is managed through an interface

## Second principle

The second principle of reusable OO design:

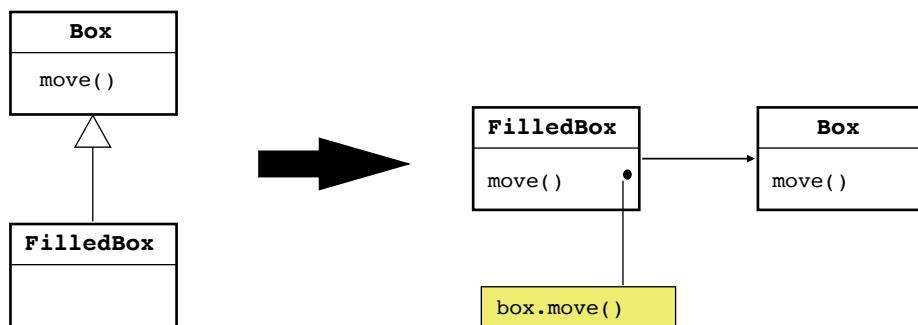


*"Favor object composition  
over class inheritance"*

## Delegation

Using composition instead of inheritance obtaining the same degree of reuse.

Remember: you can always switch from the relationship "is-a" to the relationship "has-a"



## Composition vs. inheritance

- Inheritance is powerful and convenient, but it can lead to brittle and fragile software
- It is safe to use inheritance where the developer has full control of the implementation
- It is safe to inherit from classes specially designed to be extended
- Inherit from any class, however, is dangerous

## The inheritance dangers: 1

- You inherit also the interface
  - The subclass inherits not only the implementation of some necessary methods, but it also inherits the entire interface
  - The subclass can therefore also be used by methods not foreseen for its logical interface
  - Example:

A *Stack* class that inherits from *ArrayList*, would have the advantage of using methods of a list, but would have an interface (and implementation) that would go well beyond the capabilities of the *Stack* ...

## The inheritance dangers: 2

- **The superclass can acquire new methods**
  - Suppose that a program needs to check (for security reasons) all the items inserted into a collection.  
To do this it would be enough to implement a subclass that extends all the methods of the collection, inserting the check.
  - This is good and safe only until the superclass does not implement a new insert method ...

## The inheritance dangers: 3

- **Inheritance violates the encapsulation**
  - The implementation of the superclass may change from release to release, affecting the operation of the subclasses, even if no line of code has been changed in them (and the final behavior of the superclass methods remain the same)
  - The subclass needs to know some implementation details, in order to extend the class correctly
  - **Example**

Extension of class *HashSet*

## Example: HashSet extension (1)

- Let's suppose we want to extend the class `HashSet`, adding a variable that takes into account how many items have been added to the table (without taking into account those deleted).
- The variable is incremented when calling following methods:
  - `add()`
  - `addAll()`

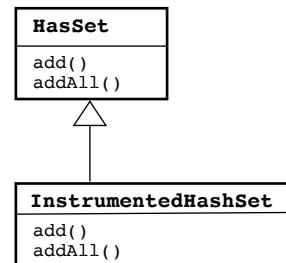
## Example: HashSet extension (2)

```
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int counter;

    public int getCounter() {
        return counter;
    }

    @Override
    public boolean add(E o) {
        counter++;
        return super.add(o);
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        counter+= c.size();
        return super.addAll(c);
    }
}
```



## Example: HashSet extension (3)

- Use

```
public class InstrumentedHashSetRun {  
  
    public static void main(String[] args) {  
        InstrumentedHashSet <String> hashSet =  
            new InstrumentedHashSet<String>();  
        hashSet.addAll(Arrays.asList("Paul", "George", "John"));  
  
        System.out.println("Counter: " + hashSet.getCounter());  
    }  
}
```

## Example: HashSet extension (4)

- We would expect 3, but the result of the counter is 6
- This is because the implementation of `addAll()`, which we call through super (inherited from `AbstractCollection`), actually uses `add()` to insert an element at a time

```
public boolean addAll(Collection<? extends E> c) {  
    boolean modified = false;  
    Iterator<? extends E> e = c.iterator();  
    while (e.hasNext()) {  
        if (add(e.next()))  
            modified = true;  
    }  
  
    return modified;  
}
```

## Example: HashSet extension (5)

- We could obviously remove the counter from `addAll()`, but this only serves to confirm that we need information on the implementation details, in order to be able to extend the class
- In addition, subsequent releases of `HashSet` could no longer use the `add()` inside `addAll()`, creating us a problem again
- The `InstrumentedHashSet` class is therefore to be considered fragile

## Solution: composition

```
public class InstrumentedHashSet<E> implements Set<E> {
    private HashSet<E> forwardSet;
    private int counter;

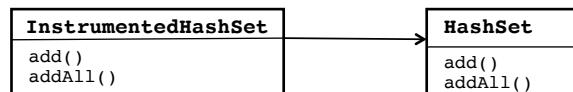
    public InstrumentedHashSet<E>(HashSet<E> forwardSet) {
        this.forwardSet = forwardSet;
    }

    public int getCounter() {
        return counter;
    }

    public boolean add(E o) {
        counter++;
        return forwardSet.add(o);
    }

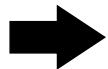
    public boolean addAll(Collection<? extends E> c) {
        counter+= c.size();
        return forwardSet.addAll(c);
    }

    ... //more "forward" methods for Set<E>
}
```



## Changes

- The key to a good reuse is to anticipate some number of changes (not all! -> refactoring)
- A design that does not consider changes complicates the program maintenance
- The design patterns help to prevent this by defining variable and independent parts, facilitating the redesign and making the system "robust" to certain types of changes
- The design patterns are often the target of a refactoring process



*"Design for change"*

## Causes of redesign

1. **Dependency between object creation and class name**  
-> Abstract Factory, Factory Method, Prototype
2. **Dependency on specific operations**  
-> Chain of responsibility, Command
3. **Dependency on software or hardware platforms**  
-> Abstract Factory, Bridge
4. **Dependency on object representations or implementations**  
-> Abstract Factory, Bridge, Memento, Proxy
5. **Algorithmic dependency**  
-> Builder, Iterator, Strategy, Template, Visitor
6. **Too narrow relationships**  
-> Abstract Factory, Bridge, Chain of responsibility, Command, Mediator, Facade, Observer
7. **Extension of functionality through hierarchy**  
-> Bridge, Chain of responsibility, Composite, Decorator, Observer, Strategy
8. **Inability to alter classes conveniently**  
-> Adapter, Decorator, Visitor

## 3 levels of reuse

### Application

- > Specific program
- > **Internal code reuse**

### Class library (toolkit)

- > Programming "for others"
- > **External code reuse**

### Framework

- > Generic programming
- > **Design reuse**

Difficulties

+

## Framework

### Definitions

- Operation based on both, abstract and concrete classes
- Connection between classes already existent at the abstract level
- Main "skeleton" of the program, to adapt and modify in order to suit your needs
  - The program flow is already available
  - The programmer creates concrete classes
  - Operation on the "Hollywood's principle" (do not call us, we will call you)
  - Generic programming
  - IoC

## Framework

### Advantages

- Specific functionality (including the program's control flow) already available, only to adapt.
- A framework represents a reusable "design"

--> **Design Patterns**

- Major advantages of maintenance, but complex implementation.

## Examples

A framework provides the basic elements of a windowing system:

- **Window**  
Scrollbar, CloseBox, Resize, ecc.
- **Mouse**  
Movements, click, tasti, ecc.

The programmer (user of the framework) sets the action to be run for each click:

- Through the definition of an appropriate function
- Through the redefinition of functions (overriding)
- ...

The Java AWT, Swing and Fx are framework examples

## Design Patterns II

### Guided Use of Design Patterns

### Discussion initial exercise

```
public class Reader {
    public static void main(String args[]) throws IOException {
        String filePath = Reader.class.getResource("Text.txt").getPath();
        BufferedReader reader = new BufferedReader(
            new FileReader(filePath));
        MyList<String> storage = new MyList<>();

        String line = reader.readLine();
        while (line != null) {
            storage.addElement(line);
            line = reader.readLine();
        }

        MyIterator<String> iter = storage.getBackwardIterator();
        while (iter.hasMoreElements()) {
            System.out.println(iter.nextElement());
        }
    }
}
```

## Discussion initial exercise

```
public class Reader {
    public static void main(String[] args) throws IOException {
        String filePath = Reader.class.getResource("Text.txt").getPath();
        File file = new File(filePath);
        MyList<String> storage = new MyList<>();

        Files.lines(file.toPath()).forEach(storage::addElement);

        MyIterator<String> iter = storage.getBackwardIterator();
        while(iter.hasMoreElements()) {
            System.out.println(iter.nextElement());
        }
    }
}
```

## Requests

### Modify the program, so that:

- the output will be written only in uppercase
- the output will be written to a file, maintaining the same reading order (forward)
- the output is followed by a statistics, where it is shown how many times the letter 'a' appears on the text

=> Try to generalize the former requests

## First and second request

```

public class Reader throws IOException {
    public static void main (String args[]){
        String fileName="Text.txt"; //args[0],
        String outFile= fileName + ".copy";
        boolean uppercase = true;
        String filePath = Reader.class.getResource(fileName).getPath();
        File file = new File(filePath);
        MyList<String> storage = new MyList<>();
        Files.lines(file.toPath()).forEach(storage::addElement);

        PrintWriter fileOut = new PrintWriter(new FileWriter(outFile));
        Iterator<String> iter = storage.getBackwardsIterator();
        String str;
        while(iter.hasMoreElements()){
            str = iter.nextElement();
            String toPrint = uppercase ? str.toUpperCase() : str;
            System.out.println(toPrint);
            fileOut.println(toPrint);
        }
    }
}

```

## Degree of OO

- Writing "ad hoc" programs is easier (and must be the first step)
- Exploiting the full potentiality of OO programming is more difficult.

But this allows:

- Better management of complex systems
- Better modularization and better use of class libraries
- Better degree of adaptation and enhanced flexibility
- Higher level of reuse-> higher productivity
- More robust systems

And it requires:

- OO design process

## Class-Responsibility-Collaboration (CRC) Cards

**Class name**

---

- responsibility 1
- responsibility 2
- ...

collaboration class

### Responsibility

- Objectives and intents of the class
- No difference between variables and methods

### Collaboration

- Classes to which some tasks are delegated

### One unique card per class

- 3 responsibilities, 4 collaborations (?)

## First design

**Reader**

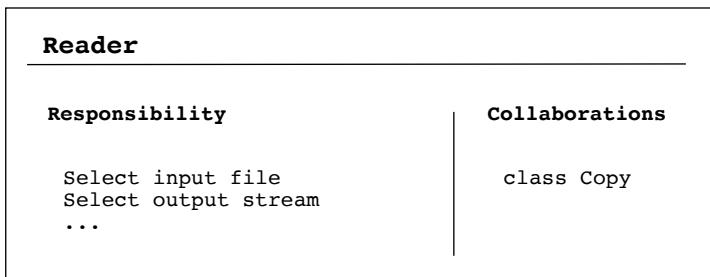
---

**Responsibilities**

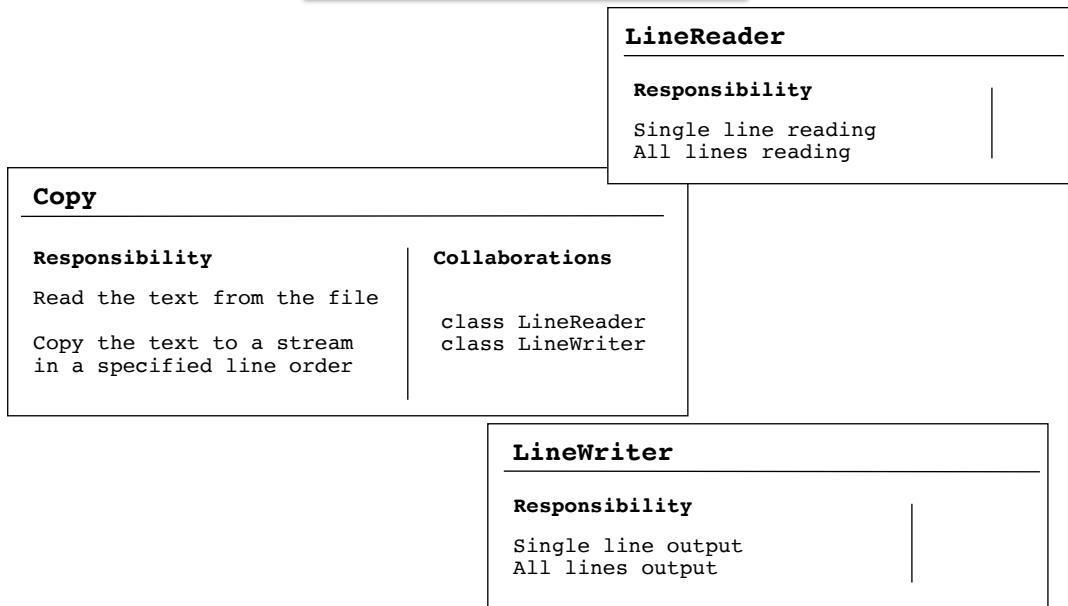
Read lines  
Store lines  
Reading  
Write lines to file  
Write lines to stdout  
Ev.  
Uppercase transformation  
Perform statistics  
Write statistics  
...

**Collaborations**

## Reviewed version (1)



## Reviewed version (2)



**Code**

```

public class Reader {
    public static void main (String args[]){
        String fileName = "Text.txt"; //args[0];
        Copy copy = new Copy(new LineReader(fileName));
        copy.toOutput(new LineWriter(System.out));
    }
}

public class Copy {
    protected MyList<String> storage;

    public Copy (LineReader in){
        storage = new MyList<String>();
        in.readAllLines(storage);
    }

    public void toOutput(LineWriter out){
        out.printAllLines(storage);
    }
    ...
}

```

**The main is used as “configuration”**

Sandro Pedrazzini

Guided Use of Design Patterns

11

**Request 1: chars conversion**

- The output (stdout or file) must be managed with the same LineWriter
- The uppercase conversion must be possible for a single output or for all outputs
- It must be possible to dynamically substitute the conversion
- The conversion algorithm must be encapsulated into an object, to which the conversion task is delegated

-> **Strategy pattern**

Sandro Pedrazzini

Guided Use of Design Patterns

12

## Strategy pattern (1)

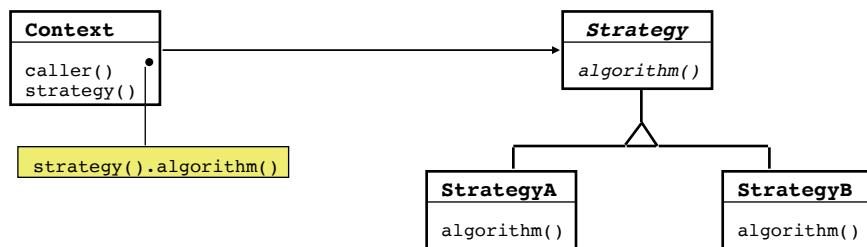
### Intent

- Define a family of interchangeable algorithms.
- The client uses one of such algorithms through a common interface

### Use

- Whenever different algorithms are appropriate, in different times
- Whenever you consider a dynamic configuration

### Structure



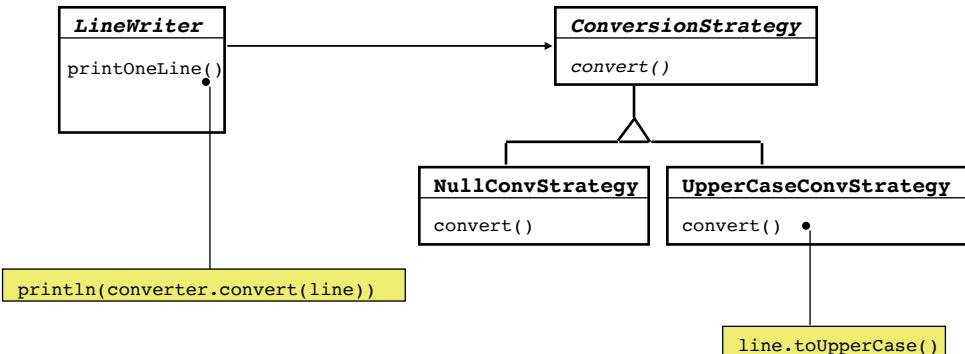
## Strategy pattern (2)

### Consequences

- Several implementations of the same algorithm are possible and can be dynamically exchanged
  - > "program to an interface, not to an implementation"
  - > "favor object composition over class inheritance"
- The class using the algorithm does not see the details of its implementation
- The concrete classes of Strategy encapsulate the data, used for the single implementation of the algorithm
- The use of Strategy helps reducing conditional checks in the code (delegated to the dispatcher)

## Strategy pattern (3)

Applied to our case



## Strategy pattern (4)

Implementation

```

public class LineWriter ... {
    private ConversionStrategy converter;

    public LineWriter(ConversionStrategy conv) {
        converter = conv;
    }

    public void printOneLine(String line) {
        println(converter.convert(line));
    }
    ...
}
  
```

```

public interface ConversionStrategy {
    convert(String source);
}
  
```

```

public class NullConverter implements ... {
    public String convert(String source) {
        return source;
    }
}
  
```

```

public class UpperCaseConverter implements ... {
    public String convert(String source) {
        return source.toUpperCase();
    }
}
  
```

## Reviewed version

### **LineReader**

#### **Responsibility**

Single line reading  
All lines reading

### **LineWriter**

#### **Responsibility**

Single line output  
All lines output

#### **Collaborations**

ConversionStrategy

### **ConversionStrategy**

#### **Responsibility**

Line conversion

## New version of *Reader*

```
public class Reader {
    public static void main (String args[]){
        String fileName = "Text.txt"; //args[0];

        Copy copy = new Copy(new LineReader(fileName));
        copy.toOutput(new LineWriter(System.out, new UppercaseConverter()));
    }
}
```

## Reader with use of Lambda

```
public class Reader {
    public static void main (String args[]){
        String fileName = "Text.txt"; //args[0];

        Copy copy = new Copy(new LineReader(fileName));
        copy.toOutput(new LineWriter(System.out, String::toUpperCase));
    }
}

or

copy.toOutput(new LineWriter(System.out, str) -> str.toUpperCase());
```

## Request 2: line order

- It must be possible to choose whether to print the lines in the original order or in the reverse order
- Use of two different implementations of the *LineWriter* abstraction:
  - *LineBackwardWriter*
  - *LineForwardWriter*
- The only difference is the iteration direction, the rest of the algorithm is identical

-> **Template Pattern**

## Template pattern (1)

### Intent

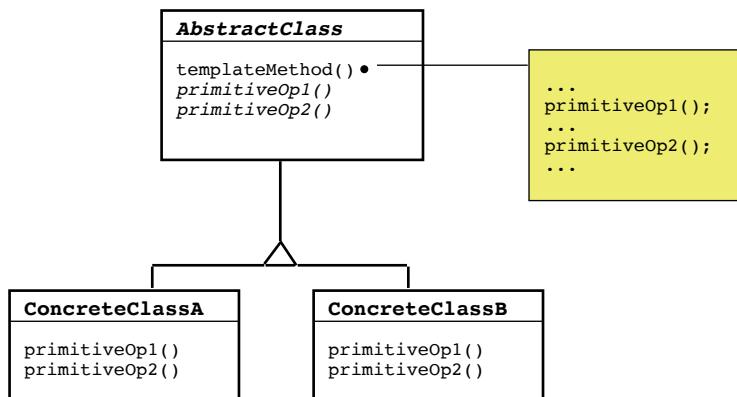
- It defines the “skeleton” of an algorithm calling operations only implemented in subclasses.
- This allows the subclasses to implement the single specific methods called by the main algorithm, adapting therefore the behavior of the latter.

### Use

- Implementation of the invariant part of an algorithm, leaving the realization of the variants elements to the subclasses
- When it is necessary to summarize a common behavior  
=> “refactoring to generalize”
- To restrict the extensibility of the algorithm to well-defined parts

## Template pattern (2)

### Structure



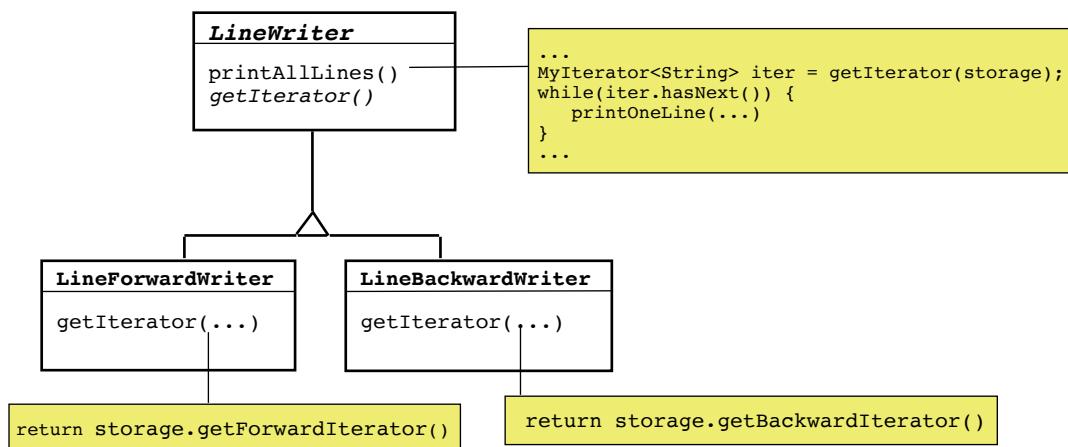
## Template pattern (3)

### Consequences

- Code reuse
- It defines the invariant behavior of the abstract class and the variant behavior of the concrete subclasses
- Global control flow and adaptability through single components  
-> framework principle
- Ability to define "hook" operations, which carry a default behavior that can be modified by subclasses

## Template pattern (4)

### Our design



## New versions of *Reader*

```
public class Reader {
    public static void main (String args[]){
        String fileName = args[0];
        String outFile  = args[0] + ".copy";

        PrintWriter fileOut = new PrintWriter(new FileWriter(outFile));

        Copy copy = new Copy(new LineReader(fileName));

        copy.toOutput(
            new LineBackwardWriter(System.out, new NullConverter()));
        copy.toOutput(
            new LineForwardWriter(fileOut, new UpperCaseConverter()));
    }
}
```

## Request 3: add statistics

- It must be possible to add statistics information to the output
- The statistics must be added dynamically, only to the desired output
- The fact of printing or not the statistics must be "transparent"
- General concept of "extension": to add more in addition to the statistics, avoiding however inheritance with a fragmentation of static subclasses.

-> **Decorator Pattern**

## Decorator pattern (1)

### Intent

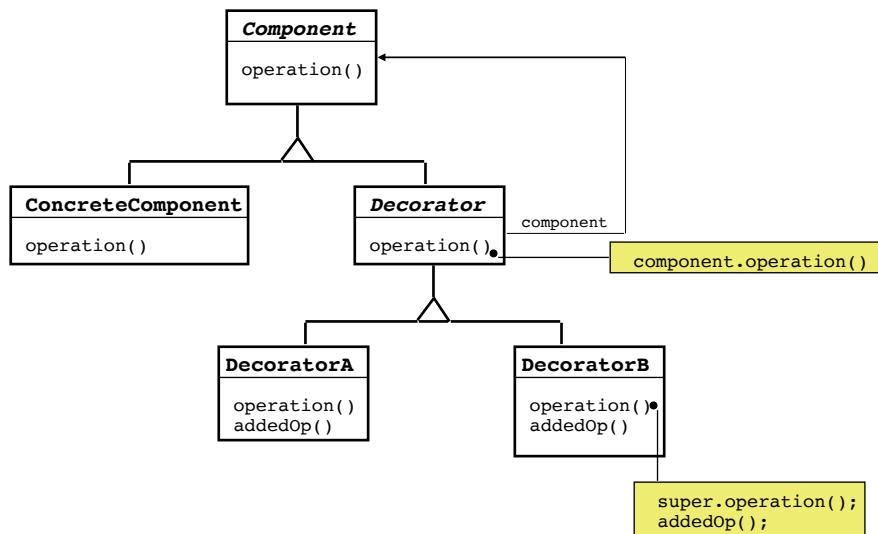
Extend single objects (not the class) with new functionalities.

### Use

- When you want to add and remove functionality, dynamically and transparently
- When the use of inheritance would lead to a too high number of subclasses
- When it is useful to recursively combine multiple decorators

## Decorator pattern (2)

### Structure



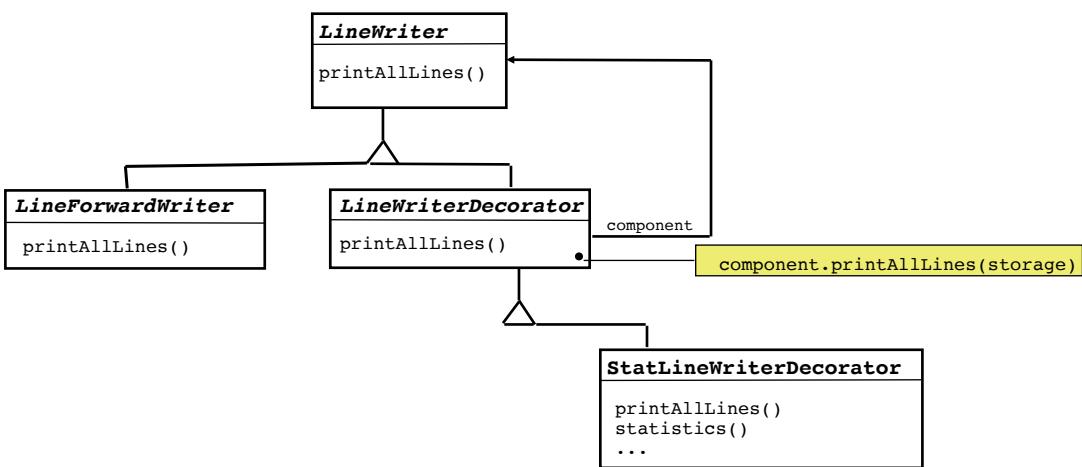
## Decorator pattern (3)

### Consequences

- The responsibilities are passed dynamically  
-> "favor object composition over class inheritance"
- Recursive combinations are possible
- Avoid a high number of subclasses
- No need to foresee and provide a superclass with many features  
-> "pay as you go" principle

## Decorator pattern (4)

### Our design



## Decorator pattern (5)

### Use of decorator in JDK: filter

#### Problem

- Several input modes (InputStreams)  
-> FileInputStream, BufferedInputStream, ...
- Various desired features and functionalities  
-> number of lines, buffer,...
- Create and combine all the possibilities through subclasses is unreasonable

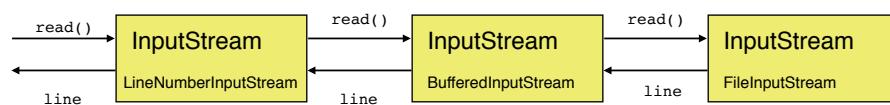
#### Solution

- Use of Decorator pattern (*Filter*), which allows any kind of combination

## Decorator pattern (6)

### Use of decorator in JDK: example

```
InputStream lnInput = new LineNumberInputStream(
    new BufferedInputStream(
        new FileInputStream(fileName)));
...
int elt = lnInput.read();
...
```



## Code (1)

```
public abstract class LineWriterDecorator implements LineWriter {
    private LineWriter lineWriter;

    public LineWriterDecorator(LineWriter lineWriter) {
        this.lineWriter = lineWriter;
    }

    public void printAllLines(MyList<String> storage) {
        lineWriter.printAllLines(storage);
    }

    public void printOneLine(String line) {
        lineWriter.printOneLine(line);
    }

    public MyIterator<String> getIterator(MyList<String> storage) {
        return lineWriter.getIterator(storage);
    }
}
```

## Code (2)

```
public class StatLineWriterDecoractor extends LineWriterDecorator {
    private char charToFind;

    public StatLineWriterDecoractor(LineWriter lineWriter, char ch) {
        super(lineWriter);
        charToFind = ch;
    }

    public void printAllLines(MyList<String> storage) {
        super.printAllLines(storage);
        makeStatistics(storage);
    }

    protected void makeStatistics(MyList<String> storage) {
        ...
    }
}
```

## Code (3)

```
public class Reader {
    public static void main (String args[]){
        String fileName = args[0];

        Copy cp = new Copy(new LineReader(fileName));

        LineWriter lw = new LineBackwardWriter(System.out,
                                              new UpperCaseConverter());

        cp.toOutput(new StatLineWriterDecorator(
                    new StatLineWriterDecorator(lw, 'a'), 'b'));
    }
}
```

## Request 4: broadcasting

- It must be possible to have an "unlimited" number of output channels
- It must be possible to dynamically add or remove new output channels, without intervening in the class code responsible for the copy
- Each single call for output should be forwarded to all output channels provided at that time

-> **Observer Pattern?** (limited version)

## Observer pattern (1)

### Intent

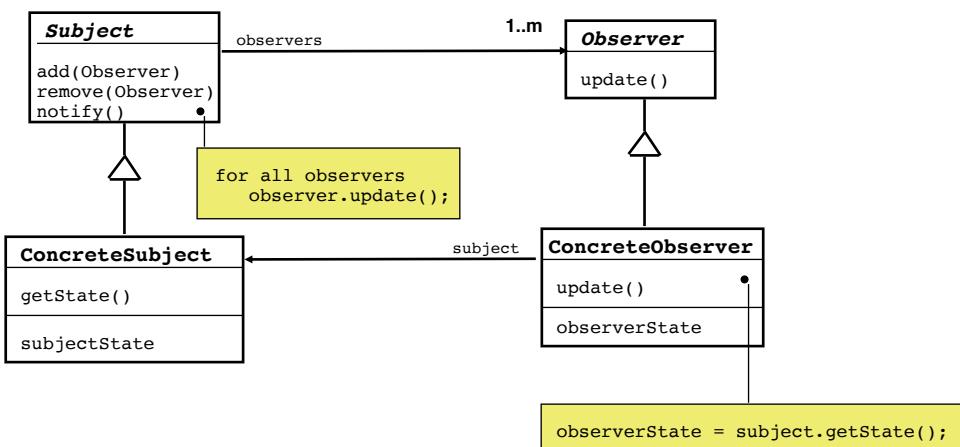
It defines a 1-m dependency among objects so that when one object changes its state, the others are informed.

### Use

- When modifying an object means also modify other ones, but you do not know how many they are
- When this dependency must be dynamic.  
An object must be able to notify its changes, without having to necessarily know the other related objects

## Observer pattern (2)

### Structure



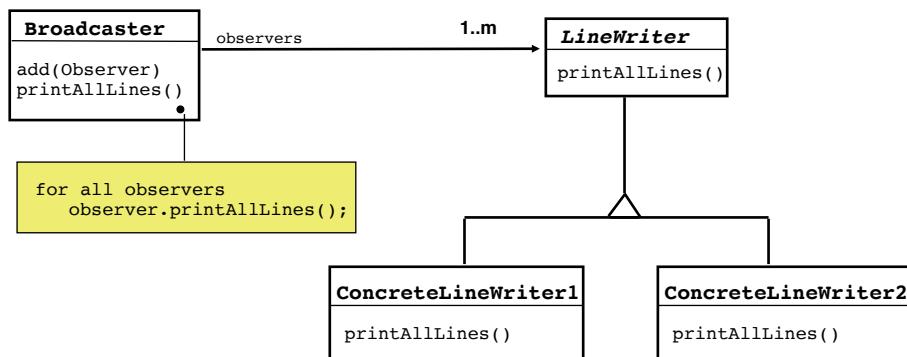
## Observer pattern (3)

### Consequences

- Observer and Subject may be changed independently  
-> "favor object composition over class inheritance"
- New observers can be dynamically added, without consequences for subject
- The Observer objects can be part of whatever class hierarchy
- The principle supports the idea of broadcasting

## Observer pattern (3)

### Our design: simplified Observer



## Observer pattern (4)

### Discussion on our case

- This is a simplified Observer version
- Exploited the implicit mechanism of "Broadcasting" (subject-observer direction)
- No need of the auto-update mechanism (observer-subject direction)
- As *Observer* abstract class we have chosen *LineWriter*, in order to ensure transparency (through polymorphism) during the *LineReader* and *LineWriter* parameter passing to *Copy*

## Code (1)

```
public class BroadCaster implements LineWriter {  
    private List<LineWriter> observers = new ArrayList<>();  
  
    public void addPrintStream(LineWriter stream) {  
        observers.add(stream);  
    }  
  
    ...  
  
    public void printAllLines(MyList<String> storage) {  
        observers.forEach((observer) -> observer.printAllLines(storage));  
    }  
}
```

**Code (2)**

```
public class Reader {
    public static void main (String args[]){
        String fileName="Text.txt", //args[0],
               outFile1= fileName + ".copy1",
               outFile2= fileName + ".copy2";

        BroadCaster bc = new BroadCaster();
        LineWriter lw0 = new LineBackwardWriter(System.out,
                                              new UpperCaseConverter());
        LineWriter lw1 = new LineBackwardWriter(outFile1,
                                              new NullConverter());
        LineWriter lw2 = new LineForwardWriter(outFile2,
                                              new NullConverter());
        bc.addPrintStream(lw0);
        bc.addPrintStream(lw1);
        bc.addPrintStream(new StatLineWriterDecorator(
                           new StatLineWriterDecorator(lw2, 'a'), 'b')));

        Copy cp = new Copy(new LineReader(fileName));
        cp.toOutput(bc);
    }
}
```

## Qualità nei processi di sviluppo

### Integrazione continua

### Integrazione continua

- Pratica dello sviluppo software
- Ogni membro del gruppo di sviluppo integra il proprio lavoro il più frequentemente possibile, solitamente almeno una volta al giorno
- Ogni integrazione viene verificata da un *build* automatico, che lancia i test e verifica eventuali problemi di integrazione

## Integrazione continua (2)

- Molti team di sviluppo trovano che una pratica del genere serve a ridurre al minimo i problemi di integrazione
- Permette al team di sviluppare software coeso in modo molto più rapido
- Aiuta a ridurre i rischi legati allo sviluppo e all'integrazione

## Motivazione

- In molti processi software l'integrazione richiede giorni, settimane o mesi
- Questo perché l'integrazione viene eseguita come ultimo passaggio prima di andare in produzione
- Per poter andare in produzione più frequentemente bisogna trasformare l'integrazione in qualcosa di automatico (integrazione vista come “non-event”)

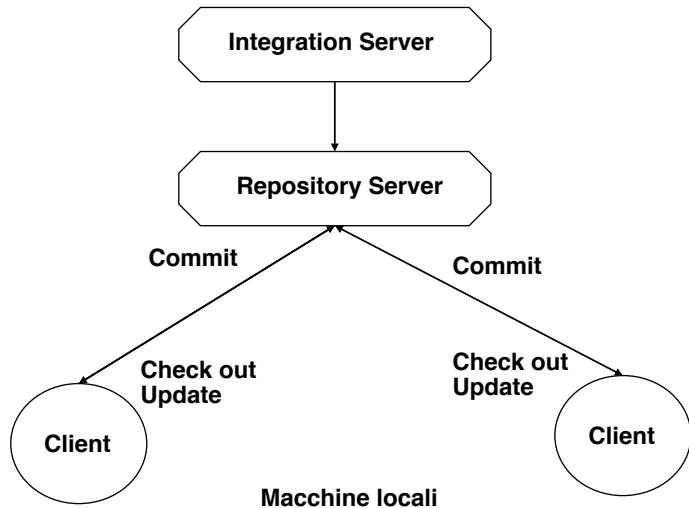
## Motivazione (2)

- Integrando regolarmente significa che ogni sviluppatore ad ogni iterazione aggiunge solo alcune ore di lavoro al progetto integrato
- Anche in questo caso (come nelle altre pratiche di XP) si tratta semplicemente di applicare in modo coerente alcune “buone abitudini”.
- Più frequentemente si integra e più l’integrazione diventa un “non-event”

## Tool

- Integrazione continua è prima di tutto una buona abitudine, perciò di base non necessitano tool particolari
- Ne esistono però alcuni (esempi: **CruiseControl**, **TeamCity**, **Jenkins**) che combinati a un sistema di gestione concorrente di sorgente (**CVS**, **Subversion**, **GIT**, ecc.) e a un sistema di build indipendente da ogni IDE (esempio: Ant, Maven, Gradle) permettono di attivare le verifiche di integrazione in modo automatico.

## Tool (2)

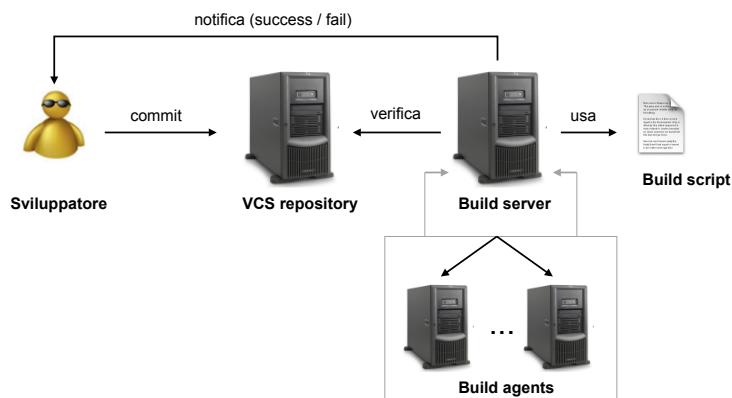


Sandro Pedrazzini

Integrazione continua

7

## Processo



Sandro Pedrazzini

Integrazione continua

8

## Tool (3)

- Esempio di build con Ant

➤ ant integrate

```
Buildfile: build.xml
clean:
all:
compile-src:
compile-tests:
integrate-db:
run-tests:
run-inspections:
delivery:
deploy:
BUILD SUCCESSFUL
Total time: 6 minutes 15 seconds
```

## Tool (4)

- Esempio di commit con Subversion (via linea di comando)

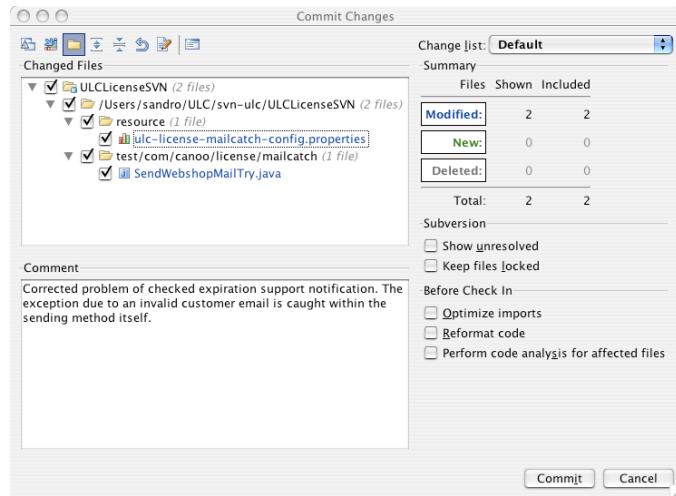
➤ svn commit -m "Aggiunta verifica della connessione"

```
Sending src/com/canoo/db/dao/DBServicesImpl.java
Transmitting file data .

Committed revision 125
```

## Tool (5)

- Esempio di commit integrato in IDE



Sandro Pedrazzini

Integrazione continua

11

## Tool (6)

- Confronto fra versioni

```

10351     mailContent.append(" no new expired support license ");
    } else {
        content.append("Expired support licenses: " + checkedSupportLicenses.size() + '\n');
        fillMailContent(mailContent, bucketsOfExpiredLicenses);
        logger.info("Send summary mail to " + baseConfig.EMAIL_ADMIN);
        MailHandler.sendSingleEmail(baseConfig.EMAIL_ADMIN, null, null, baseConfig.EMAIL_EXP_SUPPORT);
        if (baseConfig.MAIL_BE_SENT_TO_CUSTOMERS) {
            sendNotificationToSingleClients(bucketsOfExpiredLicenses);
        }
        return (SupportLicensee[]) checkedSupportLicenses.toArray(new SupportLicense[checkedSupportLicenses.size()]);
    }
}

private static void sendNotificationToSingleClients(Map expirationBuckets) throws Exception {
    if (expirationBuckets.size() == 0) {
        logger.warn("No new support license for each single client");
        Set sortedKeys = expirationBuckets.keySet();
        Iterator iter = sortedKeys.iterator();
        while (iter.hasNext()) {
            List contentList = (List) expirationBuckets.get(iter.next());
            ProductAndSupportLicensePair licensePair = (ProductAndSupportLicensePair) contentList.get(0);
            String mailCustomerText = prepareCustomerText(licensePair, contentList);
            MailHandler.sendSingleEmail(licensePair.getClientEmail(), prepareSellerAddress(licensePair));
        }
    }
}

private static String prepareSellerAddress(String secondContactEmail) {
    if (baseConfig.RESELLER_AB_CO) {
        if (secondContactEmail != null && secondContactEmail.equals("")) {
            return secondContactEmail;
        }
    }
    return null;
}

private static String prepareCustomerText(ProductAndSupportLicensePair licensePair, List contentList) {
    StringBuffer text = new StringBuffer();
    text.append("Dear ").append(licensePair.getFirstName()).append(" ").append(licensePair.getLastName());
    contentList.iterator();
    Iterator iter = contentList.iterator();
    while (iter.hasNext()) {
        ProductAndSupportLicensePair pair = (ProductAndSupportLicensePair) iter.next();
        text.append(pair.getCompanyName());
        text.append(" (").append(pair.getLicensee().getExtendedNumber()).append(")\n");
    }
    text.append(baseConfig.CUSTOMER_MAIL_TEXT);
    return text.toString();
}

private void fillBucketOfRelatedExpirationLicenses(Map expirationBucket, SupportLicensee supp
ProductLicensee) {
    RelatedProductLicenses relatedProductLicenses = dataservice.loadLicenses(supportLicense);
    for (int i = 0; i < relatedProductLicenses.size(); i++) {
        Client client = relatedProductLicenses[i].getClient();
        String key = client.getEmail() + client.getCompany() + supportLicense.getExpirationDate();
        if (client2 != null) {
            key = key + client2.getEmail() + client2.getCompany(); // singleLicense.getsize();
        }
        List contentList = (List) expirationBucket.get(key);
        if (contentList == null) {
    
```

10351 mailContent.append(" no new expired support license ");
 } else {
 content.append("Expired support licenses: " + checkedSupportLicenses.size() + '\n');
 fillMailContent(mailContent, bucketsOfExpiredLicenses);
 logger.info("Send summary mail to " + baseConfig.EMAIL\_ADMIN);
 MailHandler.sendSingleEmail(baseConfig.EMAIL\_ADMIN, null, null, baseConfig.EMAIL\_EXP\_SUPPORT);
 if (baseConfig.MAIL\_BE\_SENT\_TO\_CUSTOMERS) {
 sendNotificationToSingleClients(bucketsOfExpiredLicenses);
 }
 return (SupportLicensee[]) checkedSupportLicenses.toArray(new SupportLicense[checkedSupportLicenses.size()]);
 }
}

private static void sendNotificationToSingleClients(Map expirationBuckets) throws Exception {
 if (expirationBuckets.size() == 0) {
 logger.warn("No new support license for each single client");
 Set sortedKeys = expirationBuckets.keySet();
 Iterator iter = sortedKeys.iterator();
 while (iter.hasNext()) {
 List contentList = (List) expirationBuckets.get(iter.next());
 ProductAndSupportLicensePair licensePair = (ProductAndSupportLicensePair) contentList.get(0);
 String mailCustomerText = prepareCustomerText(licensePair, contentList);
 MailHandler.sendSingleEmail(licensePair.getClientEmail(), prepareSellerAddress(licensePair));
 }
 }
}

private static String prepareSellerAddress(String secondContactEmail) {
 if (baseConfig.RESELLER\_AB\_CO) {
 if (secondContactEmail != null && secondContactEmail.equals("")) {
 return secondContactEmail;
 }
 }
 return null;
}

private static String prepareCustomerText(ProductAndSupportLicensePair licensePair, List contentList) {
 StringBuffer text = new StringBuffer();
 contentList.iterator();
 Iterator iter = contentList.iterator();
 while (iter.hasNext()) {
 ProductAndSupportLicensePair pair = (ProductAndSupportLicensePair) iter.next();
 text.append("Dear ").append(pair.getFirstName()).append(" ").append(pair.getLastName());
 iter.next();
 StringWriter stringWriter = new StringWriter();
 content.append(message).append("\n");
 content.append(exception).append("\n");
 e.printStackTrace(stringWriter);
 MailHandler.sendSingleEmail(baseConfig.EMAIL\_ADMIN, null, null, "Problem in ulc/che
 

6 differences Deleted Changed Inserted

Sandro Pedrazzini

Integrazione continua

12

## Tool (8)

- Mail ricevuta da Cruise Control dopo il build

### BUILD COMPLETE - build.180

Date of build: 01/07/2008 16:21:50  
Time to build: 10 seconds  
Last changed: 01/07/2008 16:16:51  
Last log entry: Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Modifications since last successful build: (6)	
modified	sandro /ULCLicense/trunk/src/com/canoo/license/base/data/transaction/TransactionContext.java
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
added	sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification-3.xml
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified	sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification.xml
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified	sandro /ULCLicense/trunk/src/com/canoo/license/base/support/CheckExpiringSupport.java
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified	sandro /ULCLicense/trunk/build.xml
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified	sandro /ULCLicenseTemplates/trunk/ULCBase/ulc-support-additions.sql
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Sandro Pedrazzini

Integrazione continua

13

## Tool (9)

### Progetti in TeamCity

Welcome, sandro Logout

Projects My Changes Agents (1) Build Queue (0) Administration My Settings & Tools Configure Visible Projects

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- **CobraWunelli Maintenance Build Configuration** Idle Run ↗
  - #build.563-2 Tests passed: 159
  - No artifacts Changes (1) 10 Feb 17:56 (7m:37s)
- **CobraWunelli Trunk Build Configuration** Idle Pending (1) Run ↗
  - #build.680 Tests passed: 191
  - No artifacts Changes (1) 25 Feb 15:33 (12m:09s)

Sandro Pedrazzini

Integrazione continua

14

## Tool (10)

- History di TeamCity

The screenshot shows the TeamCity web interface for the project 'CobraWunelli'. The top navigation bar includes links for 'Projects', 'My Changes', 'Agents (1)', 'Build Queue (0)', 'Administration', 'My Settings & Tools', and a search bar. The main content area displays the 'History' tab of the build configuration 'CobraWunelli Trunk Build Configuration'. It shows a table of recent builds with columns for '#', 'Results', 'Artifacts', 'Changes', 'Started', 'Duration', 'Agent', 'Tags', and a 'Pin' icon. The table lists builds from #build.680 to #build.666. Build #680 is marked as pending for the next build. Other builds show various test results (green checkmarks, red error symbols). The bottom of the page indicates 'Showing 10 builds, see entire history'.

Sandro Pedrazzini

Integrazione continua

15

## Esempio (1)

- Supponiamo di voler aggiungere una nuova funzionalità ad un programma esistente

- Come primo passaggio devo scaricare la versione più aggiornata sulla mia macchina locale (check-out o update, nel caso avessi già una versione locale non aggiornata)
- Quando ho la versione sulla mia macchina, posso iniziare a fare le aggiunte desiderate
- Questo significa non solo aggiungere funzionalità, ma anche adattare vecchio codice, aggiungere e adattare test

Sandro Pedrazzini

Integrazione continua

16

## Esempio (2)

- Appena terminato con la nuova funzionalità e i vari test, posso creare il build sulla mia macchina locale
- Questo significa ricompilare tutto, creare e includere le varie librerie, eseguire i test
- Posso considerare di aver terminato il lavoro sulla macchina locale solo quando tutto funziona senza errori
- Ora che il build locale funziona, posso pensare di eseguire un “commit” sul repository comune

## Esempio (3)

- Non ho ancora finito: a questo punto si tratta di eseguire il build sulla macchina di integrazione
- Solo se anche questo build ha successo, si può affermare che i cambiamenti eseguiti fanno parte dell’applicazione
- Il build di integrazione può essere eseguito manualmente oppure automaticamente con tool come TeamCity, che, eseguendo un polling continuo sul repository per determinare se ci sono stati aggiornamenti, fa partire il build quando necessario

## Conflitti (1)

- Se ci sono conflitti a causa di due aggiornamenti che si accavallano, solitamente la cosa viene notata dal secondo che esegue commit
- Significa che dal suo ultimo update, qualcun altro ha eseguito un commit
- Il caso viene risolto localmente con un nuovo update (ed eventuali adattamenti) prima del commit

## Conflitti (2)

- Se il conflitto non si manifesta durante il commit (perché non sono stati toccati gli stessi file), ma esiste comunque una inconsistenza, questa viene scoperta durante il build di integrazione
- In entrambi i casi il problema viene scoperto velocemente
- La cosa più urgente da fare in questi casi è riparare il build

## Conflitti (3)

- In un ambiente di integrazione continua non si dovrebbe mai lasciare un “failed build” troppo a lungo
- Un buon team dovrebbe avere più di un build corretto al giorno
- Ognuno può quindi sviluppare a partire dall’ultima versione del build, mantenendo quindi il delta tra sviluppo e successiva integrazione il più piccolo possibile

## Elementi di CI

- Quanto visto nell’esempio precedente dimostra CI (continuous integration) nel lavoro di tutti i giorni
- Vediamo quali sono gli elementi essenziali affinché tutto questo possa avvenire in modo naturale e senza grossi problemi

## Elemento 1: Un solo repository

- I progetto software richiedono la gestione di parecchi file: utilizzare un sistema di versioning control
- Fare in modo che sia accessibile a tutti, anche da remoto
- Nel repository dovrebbe esserci tutto quanto server per eseguire il build, inclusi script, property file, librerie, ecc.
- Regola: dev'essere possibile eseguire un check out su una macchina "verGINE" ed poter subito eseguire un build

## Elemento 2: Build automatico (1)

- Build significa trasformare i sorgenti in un sistema funzionante
- Questo può essere un processo complicato che include compilazione, spostamento di file, caricamento di uno schema di DB, ecc.
- Tutto questo può essere automatizzato ed è buona cosa che lo sia, perché fa risparmiare parecchio tempo

## Elemento 2: Build automatico (2)

- Ambienti di build automatico ne esistono parecchi: Make in Unix, Ant, Maven, Gradle nel mondo Java, Nant o MSBuild per .NET, ecc.
- A dipendenza delle necessità si deve poter creare build in modo condizionale, avendo a disposizione diversi target (build con codice di test integrato, con verifiche diverse, ecc.)
- Il build può essere creato via IDE, ma dev'essere in ogni caso possibile creare un master sul server, senza IDE

## Elemento 3: Build self-testing

- Build non significa solo compilazione e link, un programma compilato correttamente può avere errori di esecuzione
- Il modo migliore per verificare il funzionamento è inserire la chiamata ai test nel processo di build
- Se un test non passa, il build deve fallire

## **Elemento 4: Si integra ogni giorno (1)**

- L'integrazione è anche un mezzo per comunicare agli altri sviluppatori quali modifiche abbiamo apportato al codice
- Integrando regolarmente si scoprono prima eventuali conflitti di sincronizzazione tra sviluppatori e si possono correggere velocemente
- Conflitti che rimangono irrisolti per giorni o settimane, sono difficili da riparare

## **Elemento 4: Si integra ogni giorno (2)**

- Regola: ogni sviluppatore dovrebbe eseguire un commit almeno una volta al giorno
- Commit frequenti (anche più volte al giorno) incoraggiano lo sviluppatore a suddividere il suo lavoro in fasi di alcune ore l'una. Questo permette di “sentire” il progredire del progetto

## Elemento 5: Mantenere il build veloce (1)

- Un elemento essenziale dell'integrazione continua è il feedback veloce
- Le “guidelines” di eXtreme Programming parlano di un massimo di 10 minuti
- Molto spesso se c’è un problema a mantenere il build a 10 minuti, questo è provocato dai test, soprattutto quelli che fanno uso di servizi esterni, come DB

## Elemento 5: Mantenere il build veloce (2)

- In caso di build troppo lunghi, si deve fare in modo di organizzare il processo a tappe (*staged build*, o *build pipeline*)
- Si parte da un “commit build” che deve durare al massimo 10 minuti, poi si possono organizzare passaggi successivi.
- Il commit build viene usato come punto di riferimento per il ciclo di integrazione continua

## Elemento 5: Mantenere il build veloce (3)

- **Esempio: two stage build**

- Il commit build si occupa della compilazione e dell'esecuzione dei test più essenziali, utilizzando oggetti "Mock" per i test sul DB
- Una volta eseguito questo (nei 10 minuti massimi) esiste un build non affidabile al 100%, ma sufficientemente sicuro da poter permettere agli sviluppatori di basare le proprie modifiche su questo ultimo build
- La seconda fase prevede l'utilizzo di tutti i test. A questo punto può anche durare alcune ore
- Se nella seconda fase si riscontrano problemi, si cerca di creare nuovi test per la prima fase che permettano di scoprire in anticipo quanto si è trovato di problematico

## Elemento 6: clonare l'ambiente di produzione (1)

- È importante poter eseguire tutti i test in un ambiente il più possibile simile a quello di produzione
- Ogni differenza può essere motivo di errore in produzione
- Clonare significa avere le stesse versioni del software, del sistema operativo, del DB, stesse librerie, stesso HW
- Ci sono dei limiti (HW troppo caro), ma spesso i vantaggi li superano ampiamente

## Elemento 6: clonare l'ambiente di produzione (2)

- Quando l'applicazione deve poter andare in produzione su ambienti diversi (esempio: applicazione desktop) è quasi impossibile provare tutte le combinazioni
- In questi casi possono però aiutare gli ambienti virtuali (VMWare, Parallels, ecc.)

## Elemento 7: Rendere accessibile il build (1)

- Uno dei maggiori problemi nello sviluppo del software è sapere se si stanno sviluppando le funzionalità realmente desiderate dal committente
- Le persone trovano più semplice vedere qualcosa di incompleto, ma che permetta di capire meglio cosa si desidera e come esprimere
- I processi agili si basano su questo e traggono vantaggio da questo tipico comportamento umano

## **Elemento 7: Rendere accessibile il build (2)**

- In queste situazioni è importante che ogni sviluppatore abbia facile accesso all'ultimo build, o al build della sua ultima iterazione
- Lo scopo è quello di poterlo usare per dimostrazioni, test, o anche unicamente per verificare cosa è cambiato negli ultimi giorni
- Il repository deve essere organizzato in modo che ci sia una chiara sequenza storica dei build

## **Elemento 8: Visione dello stato (1)**

- CI serve anche alla comunicazione, perciò è importante che ognuno possa verificare lo stato del sistema e i cambiamenti effettuati
- I programmi di CI hanno un'interfaccia Web che permette di accedere alla storia dei cambiamenti, sapere chi ha fatto le modifiche, ecc.
- Il vantaggio di un'interfaccia Web consiste nel fatto che anche sviluppatori localizzati altrove hanno facile accesso alle informazioni

## Elemento 8: Visione dello stato (2)

- Interfaccia Web di TeamCity

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- = **CobraWunelli Maintenance Build Configuration**  
#build.563-2 Tests passed: 159 | No artifacts Changes (1) 10 Feb 17:56 (7m:37s)  
Idle Run
- = **CobraWunelli Trunk Build Configuration**  
#build.680 Tests passed: 191 | No artifacts Changes (1) 25 Feb 15:33 (12m:09s)  
Idle Pending (1) Run

Sandro Pedrazzini Integrazione continua 37

## Elemento 9: Deployment automatico

- Un elemento di integrazione continua è il deployment automatico
- Questo può essere utile soprattutto se si hanno diversi deployment in ambienti diversi: automatizzare significa evitare facili fonti di errore
- Se è compresa la funzionalità di deployment in produzione, una capacità interessante è quella del rollback automatico alla versione precedente: questo permette di eliminare la “tensione” tipica del deployment

## Benefici della CI (1)

- Minor rischio
  - Ricordo di progetti senza integrazione continua: progetto terminato, ma incognita dell'integrazione
  - Difficile prevedere il tempo necessario di un'integrazione prevista solo alla fine del progetto
  - CI permette di sapere in ogni momento a che punto si è con il progetto e con gli errori

## Benefici della CI (2)

- Errori
  - CI non ci permette di eliminare gli errori, ma ci rende più semplice il compito di trovarli
  - Quando si introduce un bug nel sistema, se lo si scopre in fretta, diventa semplice anche toglierlo
  - Inoltre l'errore può essere solo introdotto in poche ore di lavoro dall'ultimo build stabile
  - Si eliminano i bug accumulati nel tempo

## Benefici della CI (3)

- Deployment più frequente
  - Grossa barriera che CI permette di eliminare
  - Permette di mostrare più rapidamente nuove features all'utente finale e ottenere di conseguenza un feedback più veloce
  - Attraverso feedback più veloce, utente e sviluppatore migliorano il loro rapporto di collaborazione
  - Si accorcia la distanza che esiste tipicamente tra sviluppatore e utente, decisiva per uno sviluppo software di successo

## Integrazione con le altre pratiche (1)

- L'utilizzo di CI si integra bene con altre pratiche di progettazione e sviluppo trattate
  - Test di unità
  - Utilizzo di standard nel codice
  - Refactoring
  - Cicli di sviluppo corti
  - Appartenenza comune del codice

## Integrazione con le altre pratiche (2)

- **Test di unità**

- Chi sviluppa, dovrebbe aggiungere codice di test al proprio codice (unit testing)
- Il test dovrebbe essere richiamato dopo ogni cambiamento
- Con CI il test viene richiamato automaticamente ad ogni build, quindi ad ogni modifica nel repository (regression test)

## Integrazione con le altre pratiche (3)

- **Utilizzo di standard**

- In ogni progetto si definiscono delle *guidelines* da seguire, in modo che l'intero codice segua gli stessi "standard"
- Spesso il controllo dell'aderenza allo standard è un processo manuale
- Con CI si può inserire una serie di analisi statiche del codice nello script di build, in grado di generare un report

## Integrazione con le altre pratiche (4)

- **Refactoring**

- Refactoring significa adattare il codice e la sua struttura interna senza modificare la funzionalità
- Uno degli scopi è quello di facilitare la manutenzione del codice
- CI assiste lo sviluppatore permettendo la chiamata di tool di inspection del codice all'interno del build, eseguito ad ogni modifica del repository

## Integrazione con le altre pratiche (5)

- **Cicli brevi**

- Significa che gli utenti devono avere a disposizione l'ultima versione del software funzionante il più spesso possibile
- CI si adatta bene a questa pratica, perché si integra più volte al giorno e ogni integrazione genera virtualmente un nuovo release
- Quando un sistema di integrazione continua è installato, un nuovo release viene generato con il minimo degli sforzi

## Integrazione con le altre pratiche (6)

- **Appartenenza comune (*collective ownership*)**
  - Ogni sviluppatore può lavorare a qualsiasi parte del sistema
  - Questo impedisce che ci sia un solo sviluppatore con conoscenze specifiche di un determinato argomento
  - CI aiuta questa pratica assicurando aderenza agli standard e richiamando continuamente i test di regressione

## Ridurre i rischi (1)

- **CI aiuta nel mantenere i rischi sotto controllo, permettendo di scoprire i problemi appena questi si manifestano**
- **Con CI e le pratiche correlate si riesce a creare una rete di qualità, che ci permette di fornire software di qualità più velocemente**

## Ridurre i rischi (2)

- Consideriamo i seguenti rischi
  - Software non deployable
  - Difetti scoperti tardi
  - Mancanza di visibilità del progetto
  - Software di bassa qualità
- Non si tratta di rendere attenti verso questi rischi (sono tutti noti), ma di permetterne la gestione

## Ridurre i rischi (3)

- Software non “deployabile”, impossibile creare il build

- Riesco a creare il build solo sulla mia macchina

In questo caso è importante sottolineare che il build dev'essere creato in modo indipendente dalla macchina, IDE utilizzato, configurazione specifica, ecc.  
È importante avere un server di integrazione, che usi uno script di build (ant) indipendente.

- Sincronizzazione con il DB

I test devono poter girare utilizzando l'ultima versione dello schema di DB. Lo schema di DB e i suoi dati minimi indispensabili devono trovarsi nel repository.  
I test devono essere in grado di eseguire un “drop” del DB e poi ricrearlo nuovo.

## Ridurre i rischi (4)

- **Difetti scoperti tardi**

- **Regression test**

Sappiamo che dobbiamo avere suite di test nel nostro progetto.  
Basta aggiungere la chiamata di questi test nello script di build, in questo modo verranno eseguiti dal server di integrazione ad ogni modifica.

- **Test coverage**

Esistono tool per verificare la percentuale di copertura dei test.  
Questi tool possono essere associati al processo di CI.

## Ridurre i rischi (5)

- **Mancanza di visibilità**

- **Informazioni**

È necessario che ogni sviluppatore nel progetto sia informato su ogni singola modifica effettuata al progetto.  
Al processo di CI può essere associata una notifica via email sulle modifiche effettuate o sugli eventuali problemi.

- **Visualizzazione grafica della struttura**

Se si desidera avere a disposizione l'ultima versione grafica della struttura del software (UML class diagram), lo si può fare inserendo nel sistema di CI la generazione a partire dall'ultima versione (esempio: Doxygen).

## Ridurre i rischi (6)

- **Software di bassa qualità (1)**

- **Aderenza del codice a standard predefiniti**

L'aderenza agli standard viene spesso controllata manualmente. In realtà esistono tool come Checkstyle, PMD o anche Sonar, che permettono di fare delle verifiche statiche del codice a partire da regole. Questi possono essere integrati al sistema CI.

- **Aderenza all'architettura**

Anche a livello di architettura ci possono essere delle guideline da seguire, ad esempio, il codice del data layer che non accede al codice del business layer, ecc.

Anche queste possono essere controllate da tool (JDepend, NDepend, ecc.) integrabili nel processo di CI.

## Ridurre i rischi (7)

- **Software di bassa qualità (2)**

- **Duplicazione del codice**

Codice duplicato rende più difficili le modifiche e la manutenzione del progetto.

È difficile scoprire dove abbiamo duplicazione di codice all'interno del progetto. Più il progetto è grande, più sviluppatori lavorano e maggiore è la probabilità di avere codice in più parti che esegue la stessa funzionalità.

Anche in questo caso esistono tool (utility CPD del tool di analisi metriche PMD, Simian, ecc.) che possono essere integrati nel processo di CI

## Come introdurre CI (1)

- Tutte le pratiche viste fin qui servono a trarre il massimo beneficio da CI
- Per iniziare non serve però applicarle tutte assieme
- Il primo passo è senza dubbio quello dell'automazione del processo di build
  - Mettiamo il progetto sotto il controllo di un sistema di gestione dei sorgenti (esempio: Subversion)
  - Facciamo in modo che con un unico comando si possa creare un build

## Come introdurre CI (2)

- Il passo successivo potrebbe essere quello di introdurre suite di test nel build automatico
- Se si hanno già test di unità nel sistema, la cosa è molto semplice, basta richiamarli durante il build
- Inizialmente il build verrà fatto partire a mano. Poi, si potrà richiamarlo automaticamente, con uno script, una volta al giorno per il nightly build
- Come ultimo passaggio, possiamo installare un server di integrazione con un software di CI (esempio: TeamCity)

# Elementi di Scrum

**Sandro Pedrazzini**

DTI-SUPSI  
sandro.pedrazzini@supsi.ch

Canoo Engineering AG  
sandro.pedrazzini@canoo.com

## Contenuto

- **Definizione**
- **Rischi**
- **Ruoli**
- **Eventi**
- **Artefatti**
- **Stima**

**Scrum**

- Scrum è un framework di lavoro per sviluppare e mantenere progetti complessi
- Rappresenta una trasformazione pratica dei concetti e delle discipline di progettazione agile



Sandro Pedrazzini

Elementi di Scrum 3

**Scrum (2)**

- Implementazione più “annacquata” delle pratiche di XP
- Abito più formale da far indossare a metodologie meno “ortodosse”
- Modo per introdurre metodologie agili in aziende in cui il loro utilizzo sembra essere problematico

Sandro Pedrazzini

Elementi di Scrum 4

## Agile Development

I like scrum. I like its simplicity.

I think that scrum's popularity is due to the fact that you can get quick business gains because it is quick to adopt

Ron Quartel, XP Coach

## Definizione

- Scrum è un framework usato per affrontare problemi complessi che richiedono adattamenti continui

- Scrum è considerato:

- Leggero
- Semplice da comprendere

**Il suo utilizzo completo richiede però esperienza**

**Definizione (2)**

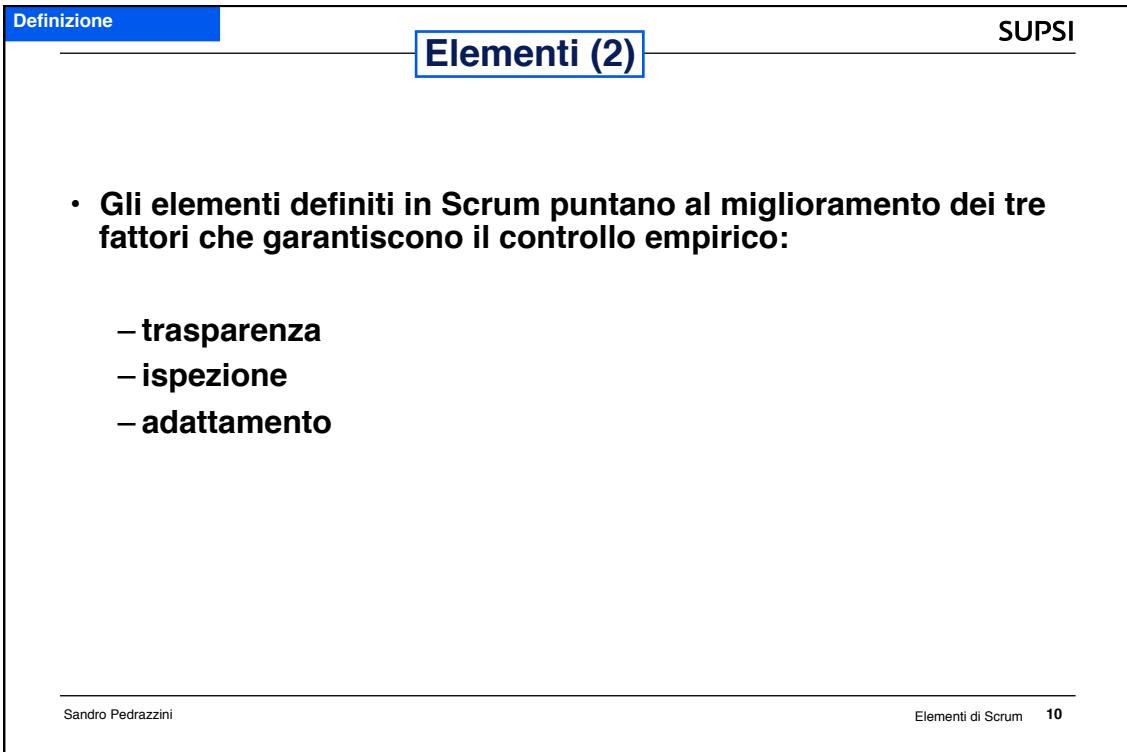
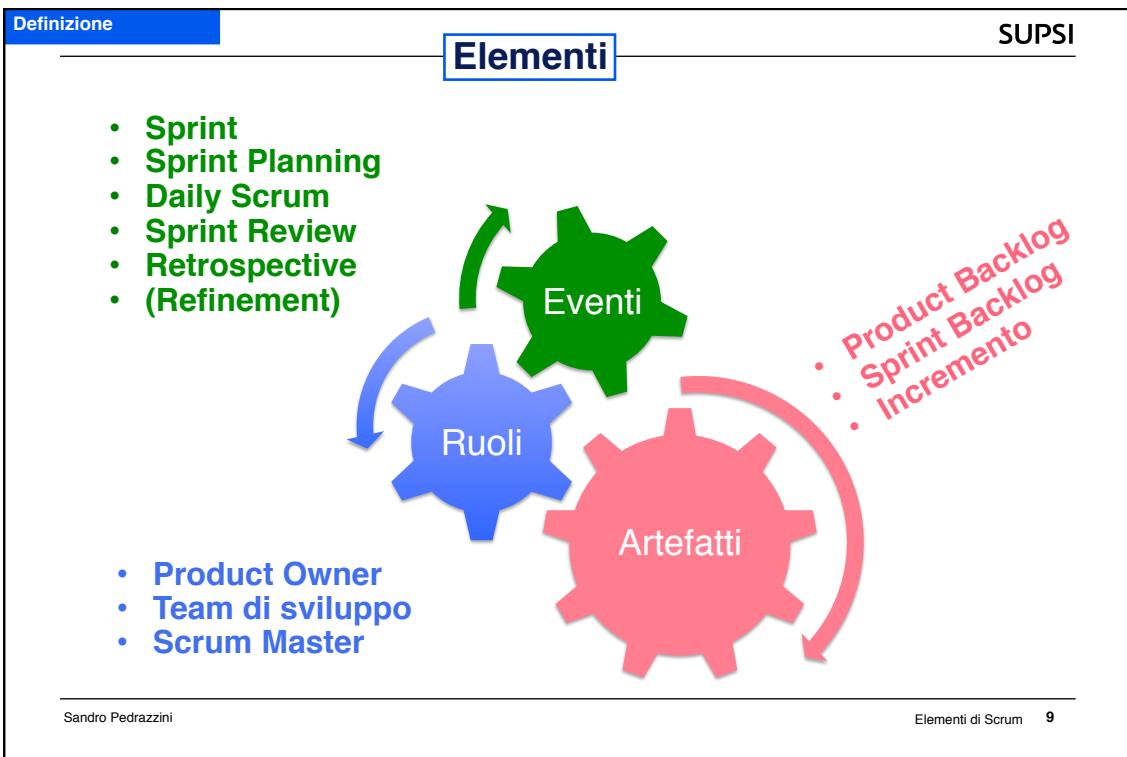
- Scrum si basa sulla teoria dei controlli empirici di processo o “empirismo”
- L’empirismo afferma che la conoscenza deriva dall’esperienza e che le decisioni si basano su ciò che si conosce
- Scrum utilizza un metodo iterativo ed un approccio incrementale per ottimizzare la prevedibilità ed il controllo del rischio.

**Definizione (3)**

- Non solo medio/piccole, ma anche grosse aziende utilizzano Scrum

According to its CIO, Apple follows a development process that is very similar to the Scrum process

DZone / Agile zone, November 2016



## Trasparenza

- La trasparenza richiede che gli aspetti significativi del processo siano definiti in modo comune e siano condivisi.

- Esempi:

La descrizione delle “story” (elementi del Product Backlog) va condivisa tra tutti i componenti del team di sviluppo.

Una definizione comune del significato “done” (fatto, completo) deve essere condivisa da chi deve eseguire il lavoro

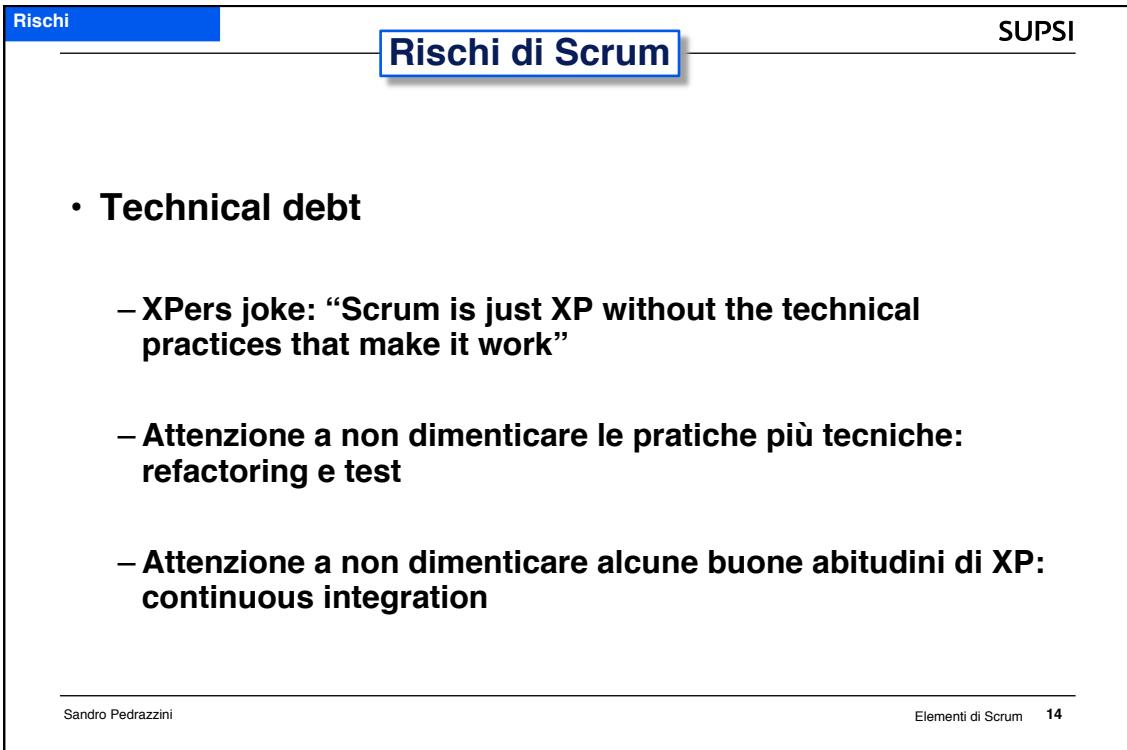
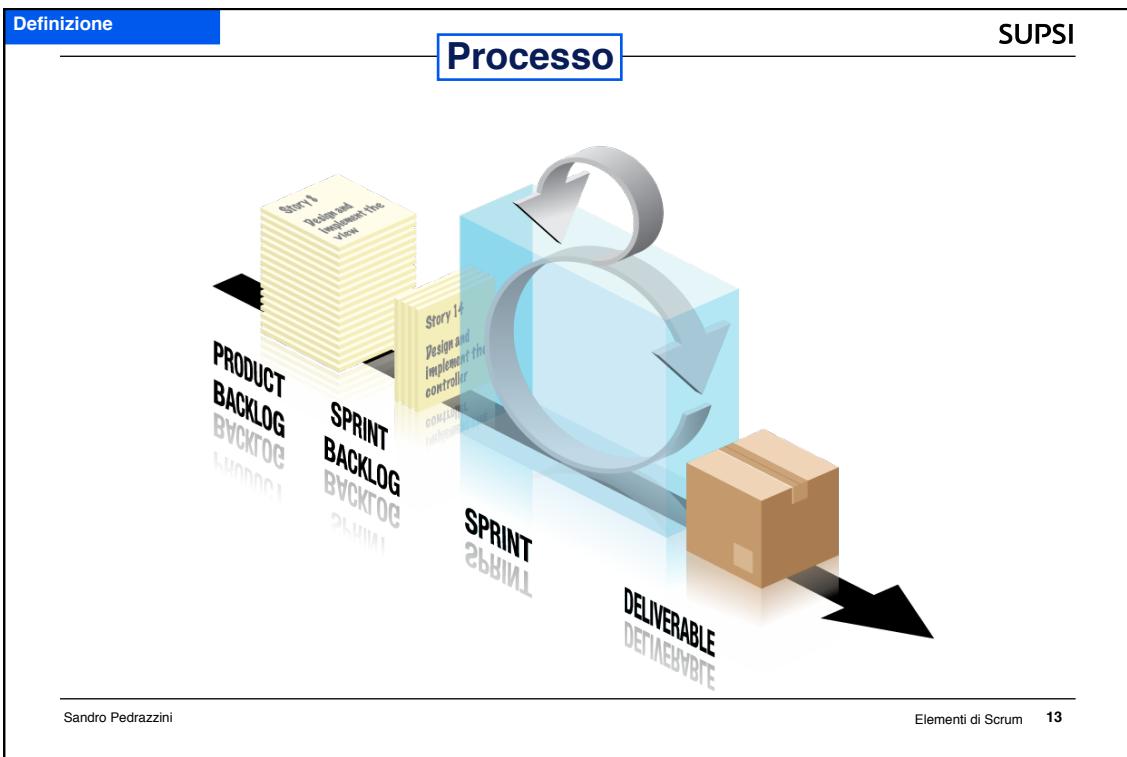
Pratiche di XP

## Ispezione / Adattamento

- Le ispezioni possono portare a una continua regolazione del processo, da applicare il più velocemente possibile
- Scrum prescrive 4-5 occasioni formali (eventi) per ispezione e adattamento:

- Sprint Planning Meeting
- Daily Scrum
- Sprint Review
- Sprint Retrospective
- + Refinement Meeting





- **Visione globale**

- **Facile dimenticare la visione globale del progetto quando si lavora a singole storie**
- **Lavorando sprint dopo sprint, senza attenzione alla qualità del codice e all'evoluzione dell'architettura, si rischia di ottenere sistemi legacy**

- **Pratiche insufficienti**

- **“Scrum is not enough”**
- **Scrum è un framework la cui implementazione / messa in pratica è lasciata al team di sviluppo**
- **Se si prova a mappare i principi che stanno alla base del manifesto agile con Scrum e XP si capisce perché è importante aggiungere a Scrum alcune pratiche di XP (Refactoring, Simple Design, Collective Code Ownership, Continuous Integration, ecc.)**

Ruoli

## Ruoli

SUPSI

- Lo Scrum team è formato da:
  - Product Owner
  - Team di sviluppo
  - Scrum Master
- Gli Scrum Team rilasciano i prodotti in modo iterativo e incrementale, massimizzando le opportunità di feedback
- I rilasci incrementali garantiscono che una versione potenzialmente utile del prodotto funzionante sia sempre disponibile

---

Sandro Pedrazzini

Elementi di Scrum 17

Ruoli

## Product Owner

SUPSI



- Il Product Owner (PO) ha la responsabilità della gestione del Product backlog (storie contenenti la descrizione dei requisiti)
- Tale gestione comporta che:
  - Gli elementi del Product Backlog siano espressi in modo chiaro
  - Il Product Backlog sia visibile e trasparente a tutti e mostri su cosa lo Scrum Team sta lavorando
  - Gli elementi del Product Backlog siano chiari al Team di Sviluppo

---

Sandro Pedrazzini

Elementi di Scrum 18

Ruoli

SUPSI

## Product Owner (2)



- Il Product Owner è una singola persona
- Il Product Owner può rappresentare nel Product Backlog desideri di un comitato, ma è lui il responsabile che questi siano visibili
- Il Product Owner fa da tramite tra stakeholder e Team di sviluppo

---

Sandro Pedrazzini

Elementi di Scrum 19

Ruoli

SUPSI

## Team di sviluppo



- Il Team di sviluppo è costituito da professionisti che lavorano per produrre un incremento potenzialmente rilasciabile di prodotto alla fine di ogni Sprint
- Soltanto i membri del Team di sviluppo creano l'incremento
- I Team di sviluppo si organizzano autonomamente al loro interno

---

Sandro Pedrazzini

Elementi di Scrum 20

Ruoli

**Team di sviluppo (2)**

SUPSI



- Sono “cross-funzionali”, con tutte le competenze come team necessarie a creare un incremento di prodotto
- Scrum non riconosce alcun titolo ai membri del Team di sviluppo al di fuori di sviluppatore
- I Team di sviluppo non contengono sotto-team dedicati a particolari domini come il testing o la business analysis.

---

Sandro Pedrazzini

Elementi di Scrum 21

Ruoli

**Team di sviluppo (3)**

SUPSI



- La dimensione ottimale del Team di Sviluppo è
  - piccola abbastanza da rimanere agile
  - grande abbastanza da avere tutte le competenze per completare il lavoro significativo

< 3 => vincoli di competenze  
 > 9 => coordinamento più oneroso

=> Nexus framework

---

Sandro Pedrazzini

Elementi di Scrum 22

Ruoli

**Scrum Master**

SUPSI



- Se il Product Owner è la figura di riferimento per ciò che riguarda il prodotto da realizzare, lo Scrum Master è il responsabile delle pratiche di Scrum
- È al servizio di Product Owner, Team di sviluppo e azienda
- Deve avere esperienza di sviluppo e di utilizzo di Scrum

Sandro Pedrazzini

Elementi di Scrum 23

Ruoli

**Scrum Master (2)**

SUPSI



- Facilita gli eventi Scrum
- Aiuta a capire e praticare l'agilità
- Fa da coach al Team di sviluppo in ambienti in cui non è ancora adottato
- Aiuta a creare gli elementi del Product Backlog in modo chiaro e conciso

Sandro Pedrazzini

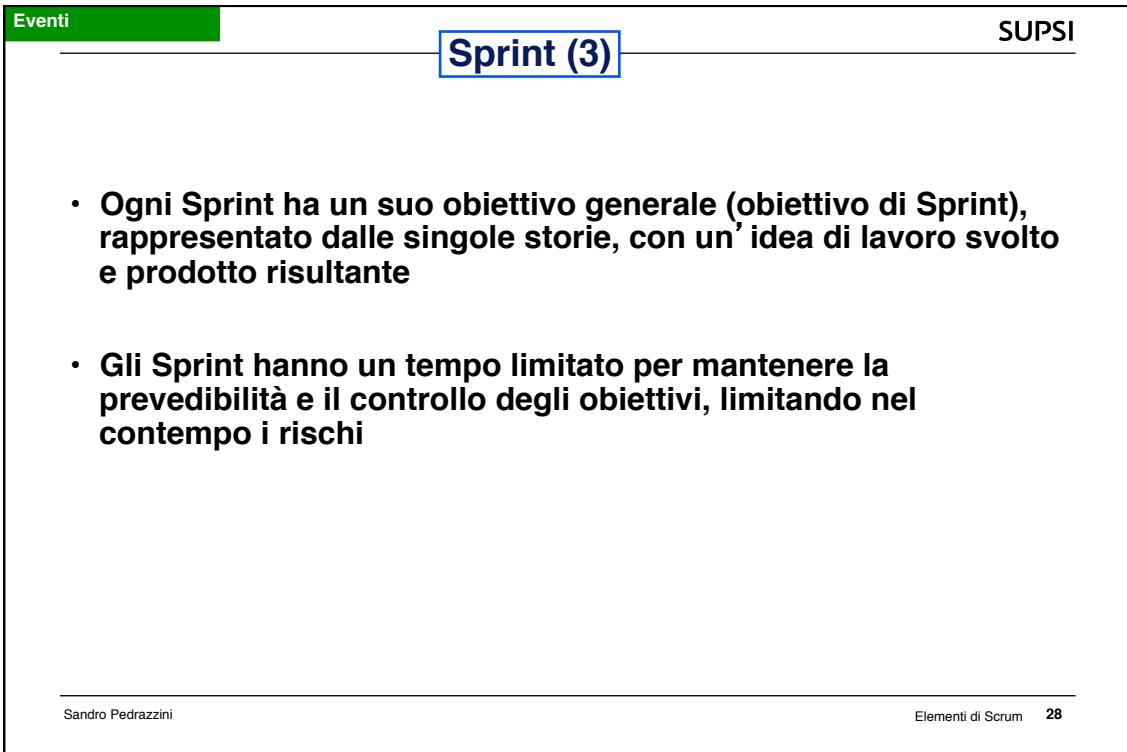
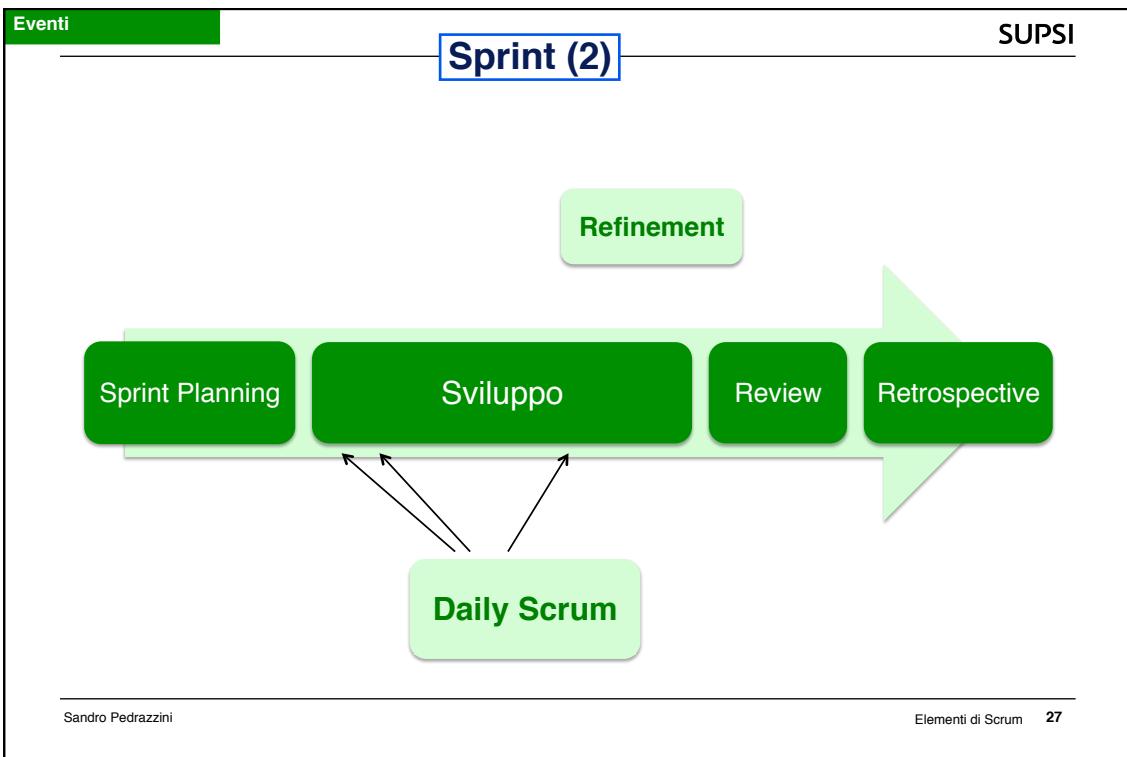
Elementi di Scrum 24

## Eventi di Scrum

- **Gli eventi previsti sono utilizzati in Scrum per creare regolarità e ridurre al minimo la necessità di riunioni non definite da Scrum stesso**
- **Gli eventi hanno un tempo massimo definito**
- **Oltre allo stesso Sprint, che è un contenitore di tutti gli altri eventi, ogni evento in Scrum è un'occasione per ispezionare e adattare qualcosa**

## Sprint

- **È l'evento di base di Scrum, rappresenta un incremento del prodotto, come somma di storie eseguite e verificate.**
- **Ha una durata che va da un paio di settimane a un massimo di un mese**
- **Un nuovo Sprint si avvia immediatamente dopo la conclusione del precedente.**



**Eventi**

**SUPSI**

## Sprint (4)

- Durante lo Sprint si affrontano le storie dello Sprint Backlog con l' obiettivo di raggiungere lo stato “accepted”**

Ready for Development (6)	In Development (1)	Ready for Testing (0)	In Testing (0)	Ready for Signoff (0)	Accepted (0)
<b>Mingle</b> #140 go to the task page via barcode reading #236 Modification of the closing trays behavior #264 Modify the research by acceptance #271 [laboratory]Wrt phase dilute #238 [Vetrino]Config Default values in vetrino and #277 Incubation status definition	#193 [vetrino]: Export the origin				

Maximize view Link to this page

Sandro Pedrazzini

Elementi di Scrum 29

**Eventi**

**SUPSI**

## Sprint (5)

**Jira**

To Do	In Progress	Done	Planned
<b>Holger Keibel</b> 3 issues <ul style="list-style-type: none"> <li><b>FINDIT-748</b> Merge all properties files into one               <ul style="list-style-type: none"> <li>Technical productizing</li> </ul> </li> <li><b>FINDIT-325</b> Make REST structure consistent               <ul style="list-style-type: none"> <li>API adjustments</li> </ul> </li> </ul>			
<b>Martin Huber</b> 3 issues <ul style="list-style-type: none"> <li><b>FINDIT-748</b> Merge all properties files into one               <ul style="list-style-type: none"> <li>Technical productizing</li> </ul> </li> <li><b>FINDIT-325</b> Make REST structure consistent               <ul style="list-style-type: none"> <li>API adjustments</li> </ul> </li> </ul>			
<b>Natalia Kulyasova</b> 6 issues <ul style="list-style-type: none"> <li><b>FINDIT-747</b> Check most recent Wik-it DB and adjust invalid category assignment tests               <ul style="list-style-type: none"> <li>Quality assurance</li> </ul> </li> <li><b>FINDIT-112</b></li> </ul>	<b>FINDIT-743</b> UI für Spezial-Definitionen: Manuell hinzugefügte/entfernte Artikel sichtbar machen <ul style="list-style-type: none"> <li>Topic Pages</li> </ul>	<b>FINDIT-511</b> Implement inline editing of multi-value fields <ul style="list-style-type: none"> <li>None</li> </ul>	<b>FINDIT-480</b> Complete the Grails app <ul style="list-style-type: none"> <li>None</li> </ul>

Holger Keibel 3 issues  
Martin Huber 3 issues  
Natalia Kulyasova 6 issues

Sandro Pedrazzini

Elementi di Scrum 30

**Eventi**

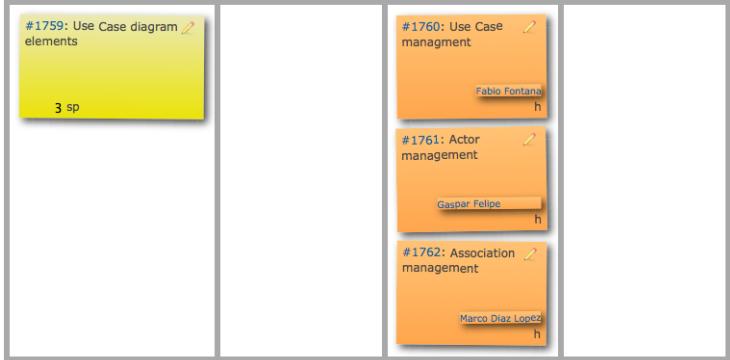
**SUPSI**

## Sprint (6)

**Redmine**

[Overview](#) [Activity](#) **Scrum** [Issues](#) [New Issue](#) [Documents](#) [Wiki](#) [Repository](#) [Settings](#)

**Sprint board**



**Sprint:** sprint-02

**Start date:** 12/10/2015

**End date:** 19/10/2015

**SPs by PBI type:** User story (3.0 s)

**Time by activity:** Total (0.0 h)

**Estimated effort:**

**Done effort:**

Sandro Pedrazzini

Elementi di Scrum 31

**Eventi**

**SUPSI**

## Sprint (7)

**Time-boxing is something that every developer using Scrum recognizes.**

**We hold firm on the amount of time in a sprint and only vary the functionality we can deliver.**

**The benefit of this is to create a predictable heartbeat of value being added to the product.**

Ken Schwaber

Sandro Pedrazzini

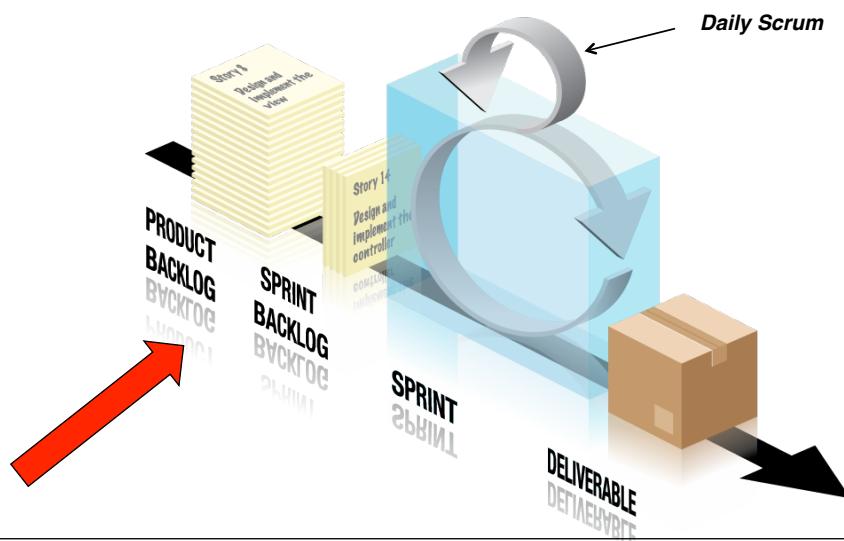
Elementi di Scrum 32

## Sprint Planning

- Si tratta di un meeting a inizio Sprint, che coinvolge l' intero team.
- Dura al massimo 8 ore per uno Sprint di un mese, 4 ore per uno Sprint di due settimane.
- Scopo è quello di determinare un obiettivo di Sprint (milestone) e scegliere quali elementi (“story”) del Product Backlog considerare per lo Sprint (Sprint Backlog)
- L' obiettivo di Sprint sarà l' elemento guida dello Sprint.

## Sprint Planning (2)

- Product Backlog e Sprint Backlog



## Sprint Planning (3)

- L'intero Team Scrum collabora nella definizione del lavoro dello Sprint
- Gli elementi da considerare sono:
  - il Product Backlog,
  - l'ultimo incremento,
  - le performance passate del team,
  - la capacità di previsione dello sforzo da parte del team
  - l'obiettivo di Sprint
- Soltanto il Team di sviluppo è in grado di valutare cosa può compiere durante lo Sprint successivo

## Sprint Planning (4)

- In una seconda fase della riunione, il Team di Sviluppo discute sugli elementi di progettazione necessari per raggiungere la realizzazione delle voci scelte per lo Sprint
- Il Product Owner può essere presente durante la seconda parte dell'incontro di Sprint Planning per chiarire le voci selezionate dal Product Backlog e per contribuire a raggiungere dei compromessi.

## Daily Scrum

- Il Daily Scrum è un evento della durata massima di 15 minuti che serve al Team di Sviluppo per sincronizzare le attività.
- Corrisponde allo “stand-up meeting” di XP
- Durante l’ incontro ogni membro del team spiega:
  - Passato
  - Futuro
  - Ostacoli



Sandro Pedrazzini

Elementi di Scrum 37

## Daily Scrum (2)

- Il meeting giornaliero, pur nella sua brevità, serve a:
  - Migliorare la comunicazione
  - Evitare altri incontri
  - Identificare e rimuovere gli ostacoli allo sviluppo
  - Evidenziare e promuovere il rapido processo decisionale
  - Migliorare il livello di conoscenza del progetto da parte del team di sviluppo
- Nell’ ottica Scrum, rappresenta un incontro chiave di ispezione e adattamento

- **Al termine dello Sprint si tiene l' incontro di Sprint Review**
- **Si tratta di un incontro tra Team di sviluppo, Product Owner e Stakeholder**
- **Serve a valutare l' incremento e adattare il Product Backlog, tenendo conto del feedback ottenuto**
- **Viene presentato l' incremento (versione attuale dell' applicazione) allo scopo di suscitare commenti e promuovere la collaborazione (per uno Sprint di 2 settimane l' incontro non deve durare più di 2 ore).**

- **La Sprint Review include i seguenti elementi:**
  - Il Team di Sviluppo mostra il lavoro che è in stato “completo” (done) e risponde alle domande sull' incremento
  - Il Product Owner identifica, attraverso la discussione, ciò che può realmente considerarsi concluso e ciò che non può esserlo (“Accepted”)
  - Il Product Owner verifica lo stato attuale del Product Backlog, ristimando il completamento, sulla base del progresso attuale.
  - L' intero gruppo collabora, così la Sprint Review fornisce un prezioso contributo alle successive riunioni di Sprint Planning.

## Sprint Retrospective

- **Occasione per il team di ispezionare sé stesso e creare un piano di miglioramento**
- **Non è necessaria dopo ogni Sprint.**
- **Si effettua dopo la Review. Si calcola un tempo proporzionale alle 3 ore per Sprint della durata di un mese.**
- **Durante ogni Sprint Retrospective, il Team Scrum pianifica i modi per aumentare la qualità del prodotto**

## Sprint Retrospective (2)

- **Gli obiettivi sono:**
  - Esaminare l'andamento dell'ultimo Sprint per quanto riguarda le persone, le relazioni, i processi e gli strumenti
  - Identificare e ordinare i maggiori elementi che sono andati bene e il potenziale di miglioramento
  - Creare un piano per attuare i miglioramenti al modo di lavorare dello Scrum Team.
- **Entro la fine della Sprint Retrospective, il Team Scrum dovrebbe aver individuato i miglioramenti da adottare nello Sprint successivo**

## Refinement Meeting (Grooming)

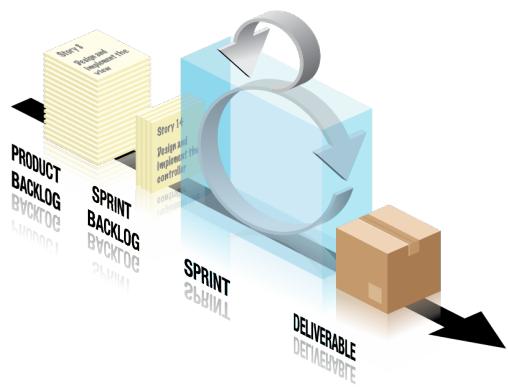
- Il Grooming è l'atto di aggiungere dettagli al Product Backlog, man mano che il lavoro avanza (backlog maintenance, attività part-time, 5-10% del tempo)
- Si tende sempre più a formalizzarlo con il Refinement Meeting, utile a preparare e stimare assieme le storie del Product Backlog, in preparazione dello Sprint Planning
- Il Team di sviluppo è responsabile di tutte le stime.

## Refinement Meeting (Grooming) (2)

- Fa da poco parte delle pratiche Scrum (GASP, “Generally Accepted Scrum Practices).
- Viene anche chiamato Backlog Grooming Meeting, cioè riunione in cui il team si assicura che il product backlog sia pronto per la scelta del prossimo sprint
- Il suo inserimento dimostra la dinamicità di SCRUM (anche la Sprint Retrospective è stata inserita come meeting in un secondo tempo)

## Gli artefatti

- **Gli artefatti definiti da Scrum sono progettati per massimizzare la trasparenza delle informazioni chiave, necessarie ad assicurare allo Scrum Team il successo nella realizzazione di un incremento.**
- **Gli artefatti sono:**
  - Product Backlog
  - Sprint Backlog
  - Incremento (deliverable)



## Product Backlog

- Il Product Backlog è un elenco ordinato di tutto ciò che potrebbe essere necessario al prodotto ed è l'unica fonte di requisiti per le modifiche da apportare al prodotto
- Il Product Owner è il responsabile del Product Backlog, compreso il suo contenuto (che va discusso con il team)

**Artefatti**

SUPSI

## Product Backlog (2)

Select: All | None Listed below: 1 to 22 of 22.

#	Name	Story Points	Epic Story
<input type="checkbox"/>	<a href="#">#37 Add new users</a>	2	#36 User management
<input type="checkbox"/>	<a href="#">#74 Enabling and password changeing</a>	3	#36 User management
<input type="checkbox"/>	<a href="#">#69 Wrong analysis</a>	3	#62 Quality control
<input type="checkbox"/>	<a href="#">#63 Ring test</a>	3	#62 Quality control
<input type="checkbox"/>	<a href="#">#108 My Future Tasks UI</a>	5	#105 Continuous UI Feedback
<input type="checkbox"/>	<a href="#">#97 Discuss and adapt side window todo list UI</a>	5	#105 Continuous UI Feedback
<input type="checkbox"/>	<a href="#">#68 Access deny</a>	5	#36 User management
<input type="checkbox"/>	<a href="#">#67 Mean time for analysis</a>	5	#66 Statistics (Part 1)
<input type="checkbox"/>	<a href="#">#7 Pair analysis</a>	5	#62 Quality control

Rank cards (22)

Pair analysis #7	training information #38	Ring test #63	Add new users #37	Mean time for analisis #67	Access deny #68	Wrong analysis #69	Automatic weight reading #55
My Future Tasks UI #108	Move all the "Info stuff in configuration #119	[micro-lab:kitchen] Different batch number when #154	[kitchen] Declare that a bottle is no more valid for a #156	TO DELETE #222	#235 [Microbiology-La Task assignment]	[vetrino]: #246 USDA quality check	[vetrino]: #249 Manage different analysis type

Sandro Pedrazzini

Elementi di Scrum 47

**Artefatti**

SUPSI

## Product Backlog (3)

- **Un Product Backlog non è mai completo. Il suo primo sviluppo definisce solo i requisiti inizialmente conosciuti e meglio compresi**
- **Il Product Backlog evolve come il prodotto e l'ambiente in cui sarà utilizzato. È dinamico e cambia continuamente per identificare ciò che serve al prodotto per essere adeguato, competitivo e utile**
- **Finché esiste un prodotto esiste anche il suo Product Backlog**

Sandro Pedrazzini

Elementi di Scrum 48

**Artefatti**

**SUPSI**

## Product Backlog (4)

- È spesso ordinato per valore, per rischio, per priorità e necessità
- Gli elementi con priorità maggiore (pronti per essere selezionati per uno Sprint) dovranno essere descritti in maggiore dettaglio e dovrà essere chiaro il loro significato di “accepted”
- Il Product Backlog è un artefatto “vivente”. I cambiamenti nei requisiti di business, le condizioni di mercato o la tecnologia possono causare cambiamenti nel Product Backlog

---

Sandro Pedrazzini

Elementi di Scrum 49

**Artefatti**

**SUPSI**

## Utilizzo delle stime

- In qualsiasi momento il lavoro totale rimanente per raggiungere un obiettivo può essere sommato.

2 (2)	3 (9)	5 (25)	8 (8)	13 (13)	21 (63)
Add new #37 users	Ring test #63  Wrong analysis #69	Pair analysis #7  Mean time #67 for analisis	training information #38	#279 [laboratory] Writing phase for weighting	Automatic weight reading #55  Change merceology #40
Enabling #74 and password changeing	Access deny #68	Discuss #97 and adapt side window todo list UI	My Future Tasks UI #108		Move all the “info stuff in configuration #119

---

Sandro Pedrazzini

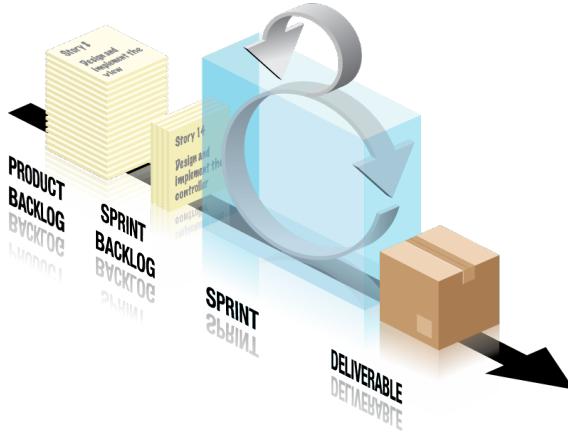
Elementi di Scrum 50

## Utilizzo delle stime (2)

- Il Product Owner traccia il lavoro totale almeno a ogni Sprint Review.
- Confronta questa cifra con il lavoro rimanente dalle precedenti Sprint Review e valuta i progressi verso il completamento dei lavori per raggiungere l'obiettivo previsto al tempo desiderato.
- Queste informazioni sono rese trasparenti a tutti gli stakeholder.

## Sprint Backlog

- Lo Sprint Backlog è l'insieme delle voci selezionate dal Product Backlog per lo Sprint, più un piano per la distribuzione dell'incremento di prodotto e la realizzazione dell'obiettivo di Sprint.



## Sprint Backlog (2)

- Lo Sprint Backlog definisce il lavoro che il Team di sviluppo eseguirà per trasformare gli elementi del Product Backlog in un incremento
- Lo Sprint Backlog rende visibile tutto il lavoro che il Team di Sviluppo identifica come necessario per soddisfare l' obiettivo di Sprint
- Rappresenta un piano di lavoro
- Aggiunte o cancellazioni allo Sprint Backlog durante lo Sprint vanno fatte unicamente dal Team di sviluppo

## Incremento del prodotto

- È chiamata “incremento” la somma di tutti gli elementi del Product Backlog completati durante gli Sprint.
- C’è un incremento di funzionalità ad ogni Sprint
- Ogni incremento è un’ aggiunta agli step precedenti

**Stima**

**Stories**

**SUPSI**

- Vale la pena investire bene nelle user stories del Product Backlog
  - Indipendenti
  - Negoziabili
  - Stimabili
  - Sufficientemente piccole
  - Testabili

---

Sandro Pedrazzini

Elementi di Scrum 55

**Stima**

**Stima**

**SUPSI**

- La stima del carico di lavoro per le singole storie del Product Backlog può essere fatta in vari modi, la cosa importante è che ci sia consistenza nella verifica e quantità stimata nei vari Sprint
- Esempi
  - Stima in ore di lavoro
  - Stima in punti (grado di difficoltà): considerare quanti punti è in grado di risolvere un Team di Sviluppo in uno Sprint

---

Sandro Pedrazzini

Elementi di Scrum 56

- **Stima in punti**

(not set) (0)	1 (2)	2 (22)	3 (63)	5 (110)	8 (88)	13 (195)	21 (126)
Process #78 modification  #213 [Microbiology-L Take out of incubator the pieces #220 in resources must be displayed as TO #222 DELETE  #231 [laboratory] right amount displayed			Modify #264 the research by acceptance	[*] #158 Server configuration  Modify #268 the way to activate the resource in		#279 [laboratory] Writing phase for weighting	

- **Con carte**



**Stima**

**SUPSI**

### Planning game (2)

- Con tabella

XS	S	M	L
S1 S2	S7	S3 S6 S5	S4

Sandro Pedrazzini

Elementi di Scrum 59

**Stima**

**SUPSI**

### Planning game (3)

- La stima definitiva serve alla preparazione dello Sprint Backlog
- Da considerare:
  - Stima delle singole storie
  - Disponibilità delle risorse
  - Buffer (adattato al team)
  - Esempio:
    - » Disponibilità risorse: 50 FTE
    - » 20% di buffer (comprendente il 10% di grooming)
    - » Sprint: composizione di storie per ca. 40 giorni

Sandro Pedrazzini

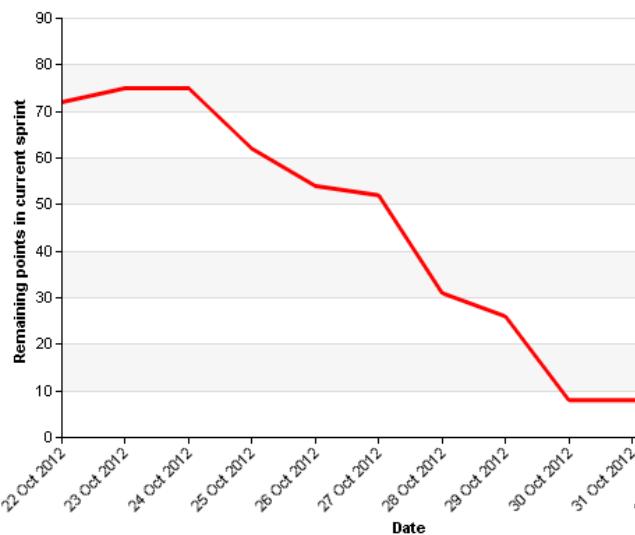
Elementi di Scrum 60

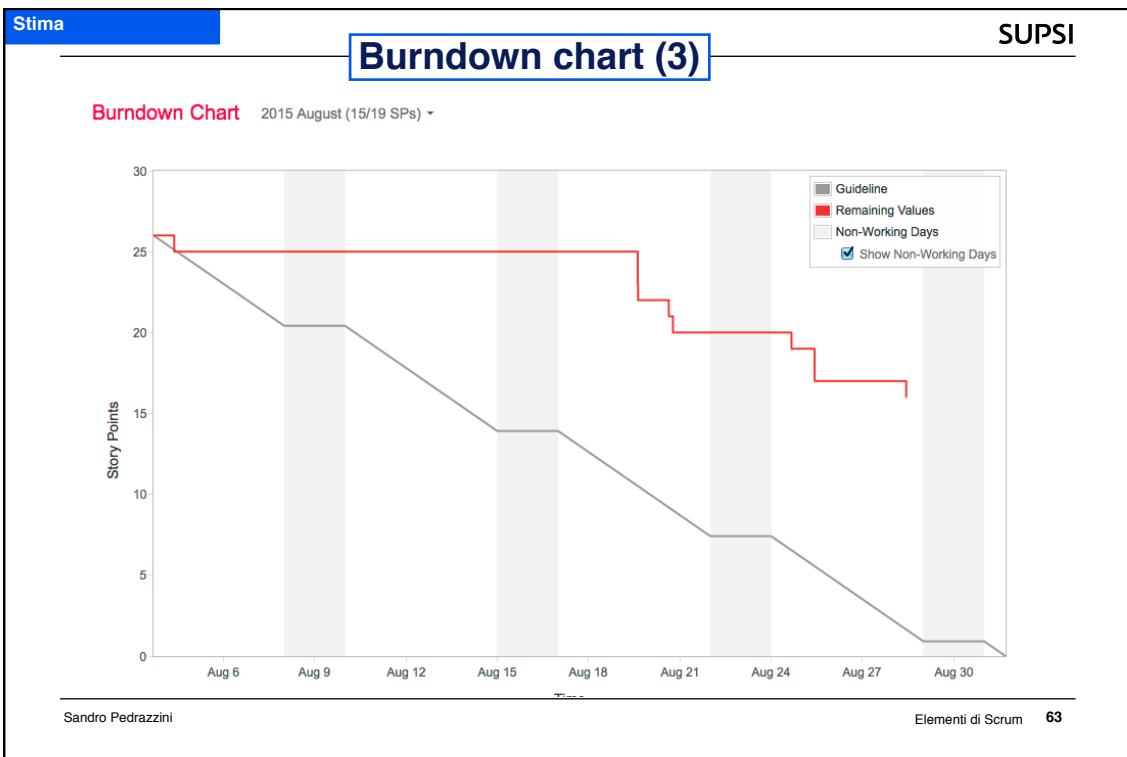
## Burndown chart

- In qualsiasi momento, durante uno Sprint, è possibile stimare il lavoro rimanente nello Sprint Backlog
- Scrum non considera il tempo impiegato per lavorare sugli elementi dello Sprint Backlog. Sono invece il lavoro rimanente e la data di fine Sprint le uniche variabili di interesse.

## Burndown chart (2)

Burndown (with points)





- Riferimenti utili** **SUPSI**
- **Cockburn Alistair:** “Agile Software Development, the Cooperative Game”
  - **Cohn Mike:** “Agile Estimating and Planning”
  - **Dräther, Koschek, Sahling:** “Scrum – kurz & gut”
  - **Eckstein Jutta:** “Agile Software Development with Distributed Teams”
  - **Kniberg Henrik:** “Scrum and Xp from the Trenches”
  - **Schwaber Ken:** “Nexus Guide”
  - **Schwaber & Sutherland:** “The Scrum Guide”
- Sandro Pedrazzini Elementi di Scrum 64

# **Software Engineering: Scrum**

## **1. Preparazione Backlog**

Questa parte di esercizio serve a spiegare il progetto e a creare il backlog di prodotto.

Organizzarsi in team da 4-6 persone e disporsi in modo tale da avere un tavolo a disposizione per ogni team. Una persona fa da product owner, cioè coordina la fase di descrizione delle storie (per semplicità un solo livello), gli altri contribuiscono alla discussione e all'eventuale aggiunta di dettagli realizzativi.

Il progetto richiede la costruzione degli edifici di un piccolo villaggio, usando gli elementi basilari dei mattoncini LEGO (virtualmente a disposizione: mattoncini di base di varie dimensioni, serramenti, ruote, ecc. Non disponibili: elementi complessi preconfezionati).

Attenzione: il progetto consiste nella realizzazione in LEGO, NON si tratta della simulazione di una costruzione reale.

Elementi che caratterizzano la città:

- 3 case unifamiliari (1 piano, dimensioni della base: ca. 20 cm x 30 cm, altezza di un piano ca. 8 cm. Gli altri oggetti andranno tutti realizzati in scala con questi)
- 3 case unifamiliari di 2 piani
- 2 case bifamiliari a schiera
- 2 palazzi da 10 appartamenti ognuno
- scuola
- asilo
- casa comunale
- chiesa
- negozio di quartiere
- palazzo da adibire a uffici (3 piani)
- ufficio postale
- ponte su fiume
- stazione ferroviaria

Strade, ferrovia, piazze, parchi, ecc. non vanno considerati nella realizzazione.

1. Preparare una prima versione di backlog utilizzando le informazioni a disposizione. Su ogni foglietto andrà descritto cosa si intende realizzare e almeno abbozzata l'idea di base del singolo progetto.
2. L'intero team fa passare e legge le singole descrizioni (grooming), aggiungendo eventuali informazioni che ritiene utili per la specificazione degli elementi da realizzare, a parole o con schizzi.

## 2. Stima del tempo

Accordarsi in gruppo sul concetto condiviso di “done” valido per tutte le storie del backlog.

In seguito, dopo aver verificato che tutti sono in chiaro su quanto va realizzato, disporre individualmente i foglietti su 4 colonne, rappresentanti i valori S, M, L e XL.

S: storie il cui tempo realizzativo arriva al massimo a 15 minuti.

M: storie tra i 15 e 30 minuti.

L: storie tra 30 e 60 minuti

XL: storie superiori all'ora di lavoro

Ognuno si occupa di disporre una parte di foglietti.

Quando tutte le storie sono state disposte sulle colonne, ogni membro del team verifica le posizioni di quelle trattate dai colleghi, identificando le storie che a suo parere andrebbero inserite in colonne diverse.

La posizione di ogni storia identificata andrà in seguito discussa con il resto del team, fino ad arrivare ad un accordo.

Al termine di questa fase, decidere se è il caso di assegnare valori di stima in tempo più precisi alle singole storie (in ogni caso andrà fatto per eventuali storie XL, per le altre si può considerare il valore massimo).

## 3. Backlog del primo sprint

Scopo di questa parte di esercizio è quello di organizzare uno sprint.

Per il primo sprint abbiamo a disposizione un tempo complessivo di 5 ore/uomo (somma totale delle risorse).

Identificare l'obiettivo dello sprint (sprint goal) e preparare il backlog di sprint considerando: risorse disponibili, grooming, daily load e storie a disposizione, con loro eventuali vincoli.

Per le storie del backlog di sprint, verificare se è necessario definire concetti di “done” specifici, che vadano oltre quanto definito in precedenza in modo più generico.

## 4. Consegne

Ogni gruppo dovrà consegnare i seguenti elementi:

- Documento con la lista delle storie individuate (titoli) e con la stima per le singole storie.
- Stima totale del progetto.
- Descrizione del primo Backlog di Sprint (obiettivo di sprint, storie identificate, stima).

## Qualità nei processi di sviluppo

### Integrazione continua

### Integrazione continua

- Pratica dello sviluppo software
- Ogni membro del gruppo di sviluppo integra il proprio lavoro il più frequentemente possibile, solitamente almeno una volta al giorno
- Ogni integrazione viene verificata da un *build* automatico, che lancia i test e verifica eventuali problemi di integrazione

## Integrazione continua (2)

- Molti team di sviluppo trovano che una pratica del genere serve a ridurre al minimo i problemi di integrazione
- Permette al team di sviluppare software coeso in modo molto più rapido
- Aiuta a ridurre i rischi legati allo sviluppo e all'integrazione

## Motivazione

- In molti processi software l'integrazione richiede giorni, settimane o mesi
- Questo perché l'integrazione viene eseguita come ultimo passaggio prima di andare in produzione
- Per poter andare in produzione più frequentemente bisogna trasformare l'integrazione in qualcosa di automatico (integrazione vista come “non-event”)

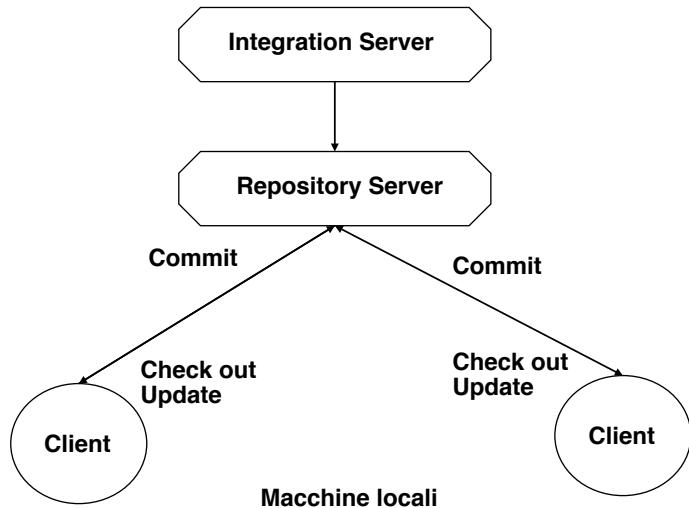
## Motivazione (2)

- Integrando regolarmente significa che ogni sviluppatore ad ogni iterazione aggiunge solo alcune ore di lavoro al progetto integrato
- Anche in questo caso (come nelle altre pratiche di XP) si tratta semplicemente di applicare in modo coerente alcune “buone abitudini”.
- Più frequentemente si integra e più l’integrazione diventa un “non-event”

## Tool

- Integrazione continua è prima di tutto una buona abitudine, perciò di base non necessitano tool particolari
- Ne esistono però alcuni (esempi: **CruiseControl**, **TeamCity**, **Jenkins**) che combinati a un sistema di gestione concorrente di sorgente (**CVS**, **Subversion**, **GIT**, ecc.) e a un sistema di build indipendente da ogni IDE (esempio: Ant, Maven, Gradle) permettono di attivare le verifiche di integrazione in modo automatico.

## Tool (2)

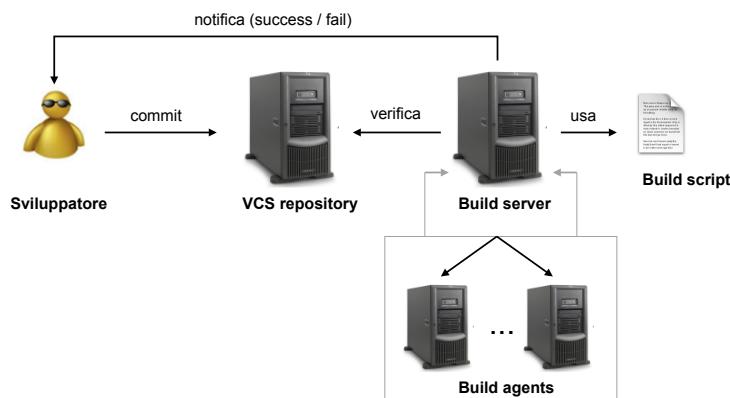


Sandro Pedrazzini

Integrazione continua

7

## Processo



Sandro Pedrazzini

Integrazione continua

8

## Tool (3)

- Esempio di build con Ant

➤ ant integrate

```
Buildfile: build.xml
clean:
all:
compile-src:
compile-tests:
integrate-db:
run-tests:
run-inspections:
delivery:
deploy:
BUILD SUCCESSFUL
Total time: 6 minutes 15 seconds
```

## Tool (4)

- Esempio di commit con Subversion (via linea di comando)

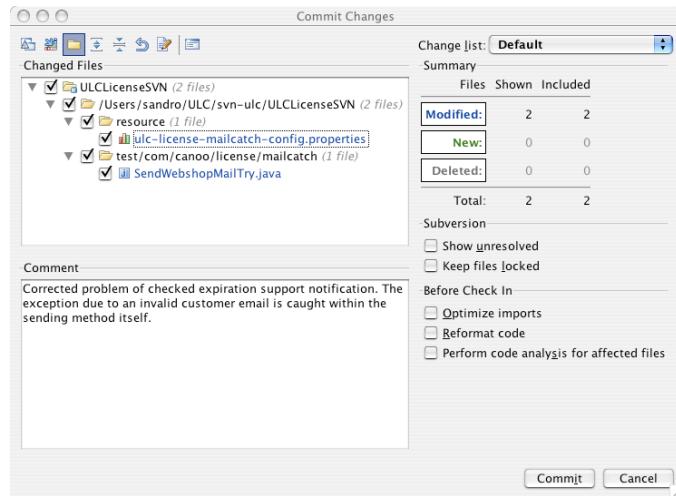
➤ svn commit -m "Aggiunta verifica della connessione"

```
Sending src/com/canoo/db/dao/DBServicesImpl.java
Transmitting file data .

Committed revision 125
```

## Tool (5)

- Esempio di commit integrato in IDE



Sandro Pedrazzini

Integrazione continua

11

## Tool (6)

- Confronto fra versioni

```

10351     mailContent.append(" no new expired support license ");
    } else {
        content.append("Expired support licenses: " + checkedSupportLicenses.size() + "\n");
        fillMailContent(mailContent, bucketsOfExpiredLicenses);
        logger.info("Send summary mail to " + baseConfig.EMAIL_ADMIN);
        MailHandler.sendEmail(baseConfig.EMAIL_ADMIN, null, null, baseConfig.EMAIL_EXP_SUPPORT);
        if (baseConfig.MAIL_BE_SENT_TO_CUSTOMER) {
            sendNotificationToSingleClients(bucketsOfExpiredLicenses);
        }
        return (SupportLicensee[]) checkedSupportLicenses.toArray(new SupportLicense[checkedSupportLicenses.size()]);
    }
}

private static void sendNotificationToSingleClients(Map expirationBuckets) throws Exception {
    if (expirationBuckets.size() == 0) {
        logger.warn("No new support license for each single client");
        Set sortedKeys = expirationBuckets.keySet();
        Iterator iter = sortedKeys.iterator();
        while (iter.hasNext()) {
            List contentList = (List) expirationBuckets.get(iter.next());
            ProductAndSupportLicensePair licensePair = (ProductAndSupportLicensePair) contentList.get(0);
            String mailCustomerText = prepareCustomerText(licensePair, contentList);
            MailHandler.sendSimpleMessage(licensePair.getClientEmail(), prepareSellerAddress(licensePair));
        }
    }
}

private static String prepareSellerAddress(String secondContactEmail) {
    if (baseConfig.RESELLER_AB_CO) {
        if (secondContactEmail != null && secondContactEmail.equals("")) {
            return secondContactEmail;
        }
    }
    return null;
}

private static String prepareCustomerText(ProductAndSupportLicensePair licensePair, List contentList) {
    StringBuffer text = new StringBuffer();
    text.append("Dear ").append(licensePair.getFirstName()).append(" ").append(licensePair.getLastName());
    contentList.iterator();
    Iterator iter = contentList.iterator();
    while (iter.hasNext()) {
        ProductAndSupportLicensePair pair = (ProductAndSupportLicensePair) iter.next();
        text.append(pair.getCompanyName());
        text.append(" (").append(pair.getLicensee().getExtendedNumber()).append(")\n");
    }
    text.append(baseConfig.CUSTOMER_MAIL_TEXT);
    return text.toString();
}

private void fillBucketOfRelatedExpirationLicenses(Map expirationBucket, SupportLicensee supp
ProductLicensee) {
    RelatedProductLicenses relatedProductLicenses = dataservice.loadLicenses(supportLicense);
    for (int i = 0; i < relatedProductLicenses.size(); i++) {
        Client client = relatedProductLicenses[i].getClient();
        String key = client.getEmail() + client.getCompany() + supportLicense.getExpirationDate();
        if (client2 != null) {
            key = key + client2.getEmail() + client2.getCompany(); // singleLicense.getsize();
        }
        List contentList = (List) expirationBucket.get(key);
        if (contentList == null) {
    
```

6 differences Deleted Changed Inserted

Sandro Pedrazzini

Integrazione continua

12

## Tool (8)

- Mail ricevuta da Cruise Control dopo il build

### BUILD COMPLETE - build.180

Date of build: 01/07/2008 16:21:50  
Time to build: 10 seconds  
Last changed: 01/07/2008 16:16:51  
Last log entry: Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Modifications since last successful build: (6)

modified sandro /ULCLicense/trunk/src/com/canoo/license/base/data/transaction/TransactionContext.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
added sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification-3.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/src/com/canoo/license/base/support/CheckExpiringSupport.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/build.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicenseTemplates/trunk/ULCBase/ulc-support-additions.sql	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Sandro Pedrazzini

Integrazione continua

13

## Tool (9)

### Progetti in TeamCity

Welcome, sandro Logout

Projects My Changes Agents (1) Build Queue (0) Administration My Settings & Tools Configure Visible Projects

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- **CobraWunelli Maintenance Build Configuration** Idle Run
- #build.563-2 Tests passed: 159
- No artifacts Changes (1) 10 Feb 17:56 (7m:37s)
- **CobraWunelli Trunk Build Configuration** Idle Pending (1) Run
- #build.680 Tests passed: 191
- No artifacts Changes (1) 25 Feb 15:33 (12m:09s)

Sandro Pedrazzini

Integrazione continua

14

## Tool (10)

- History di TeamCity

The screenshot shows the TeamCity web interface for the project 'CobraWunelli'. The top navigation bar includes 'Projects', 'My Changes', 'Agents (1)', 'Build Queue (0)', 'Administration', 'My Settings & Tools', and a search bar. The main content area displays the 'History' tab of the build configuration 'CobraWunelli Trunk Build Configuration'. It shows a table of recent builds with columns for '#', 'Results', 'Artifacts', 'Changes', 'Started', 'Duration', 'Agent', 'Tags', and a 'Pin' icon. The table lists builds from #build.680 to #build.666. Build #680 is green (Tests passed: 191). Builds #678, #677, #676, #672, #671, #670, and #669 are green (Tests passed: 185, 172, 172, 175, 175, 175, 175). Build #668 is red (Tests failed: 1 (1 new), passed: 171). Build #666 is red ([Execution timeout] Tests passed: 171). The status bar at the bottom indicates 'Integrazione continua' and the page number '15'.

## Esempio (1)

- Supponiamo di voler aggiungere una nuova funzionalità ad un programma esistente

- Come primo passaggio devo scaricare la versione più aggiornata sulla mia macchina locale (check-out o update, nel caso avessi già una versione locale non aggiornata)
- Quando ho la versione sulla mia macchina, posso iniziare a fare le aggiunte desiderate
- Questo significa non solo aggiungere funzionalità, ma anche adattare vecchio codice, aggiungere e adattare test

## Esempio (2)

- Appena terminato con la nuova funzionalità e i vari test, posso creare il build sulla mia macchina locale
- Questo significa ricompilare tutto, creare e includere le varie librerie, eseguire i test
- Posso considerare di aver terminato il lavoro sulla macchina locale solo quando tutto funziona senza errori
- Ora che il build locale funziona, posso pensare di eseguire un “commit” sul repository comune

## Esempio (3)

- Non ho ancora finito: a questo punto si tratta di eseguire il build sulla macchina di integrazione
- Solo se anche questo build ha successo, si può affermare che i cambiamenti eseguiti fanno parte dell’applicazione
- Il build di integrazione può essere eseguito manualmente oppure automaticamente con tool come TeamCity, che, eseguendo un polling continuo sul repository per determinare se ci sono stati aggiornamenti, fa partire il build quando necessario

## Conflitti (1)

- Se ci sono conflitti a causa di due aggiornamenti che si accavallano, solitamente la cosa viene notata dal secondo che esegue commit
- Significa che dal suo ultimo update, qualcun altro ha eseguito un commit
- Il caso viene risolto localmente con un nuovo update (ed eventuali adattamenti) prima del commit

## Conflitti (2)

- Se il conflitto non si manifesta durante il commit (perché non sono stati toccati gli stessi file), ma esiste comunque una inconsistenza, questa viene scoperta durante il build di integrazione
- In entrambi i casi il problema viene scoperto velocemente
- La cosa più urgente da fare in questi casi è riparare il build

## Conflitti (3)

- In un ambiente di integrazione continua non si dovrebbe mai lasciare un “failed build” troppo a lungo
- Un buon team dovrebbe avere più di un build corretto al giorno
- Ognuno può quindi sviluppare a partire dall’ultima versione del build, mantenendo quindi il delta tra sviluppo e successiva integrazione il più piccolo possibile

## Elementi di CI

- Quanto visto nell’esempio precedente dimostra CI (continuous integration) nel lavoro di tutti i giorni
- Vediamo quali sono gli elementi essenziali affinché tutto questo possa avvenire in modo naturale e senza grossi problemi

## Elemento 1: Un solo repository

- I progetto software richiedono la gestione di parecchi file: utilizzare un sistema di versioning control
- Fare in modo che sia accessibile a tutti, anche da remoto
- Nel repository dovrebbe esserci tutto quanto server per eseguire il build, inclusi script, property file, librerie, ecc.
- Regola: dev'essere possibile eseguire un check out su una macchina "verGINE" ed poter subito eseguire un build

## Elemento 2: Build automatico (1)

- Build significa trasformare i sorgenti in un sistema funzionante
- Questo può essere un processo complicato che include compilazione, spostamento di file, caricamento di uno schema di DB, ecc.
- Tutto questo può essere automatizzato ed è buona cosa che lo sia, perché fa risparmiare parecchio tempo

## Elemento 2: Build automatico (2)

- Ambienti di build automatico ne esistono parecchi: Make in Unix, Ant, Maven, Gradle nel mondo Java, Nant o MSBuild per .NET, ecc.
- A dipendenza delle necessità si deve poter creare build in modo condizionale, avendo a disposizione diversi target (build con codice di test integrato, con verifiche diverse, ecc.)
- Il build può essere creato via IDE, ma dev'essere in ogni caso possibile creare un master sul server, senza IDE

## Elemento 3: Build self-testing

- Build non significa solo compilazione e link, un programma compilato correttamente può avere errori di esecuzione
- Il modo migliore per verificare il funzionamento è inserire la chiamata ai test nel processo di build
- Se un test non passa, il build deve fallire

## **Elemento 4: Si integra ogni giorno (1)**

- L'integrazione è anche un mezzo per comunicare agli altri sviluppatori quali modifiche abbiamo apportato al codice
- Integrando regolarmente si scoprono prima eventuali conflitti di sincronizzazione tra sviluppatori e si possono correggere velocemente
- Conflitti che rimangono irrisolti per giorni o settimane, sono difficili da riparare

## **Elemento 4: Si integra ogni giorno (2)**

- Regola: ogni sviluppatore dovrebbe eseguire un commit almeno una volta al giorno
- Commit frequenti (anche più volte al giorno) incoraggiano lo sviluppatore a suddividere il suo lavoro in fasi di alcune ore l'una. Questo permette di “sentire” il progredire del progetto

## Elemento 5: Mantenere il build veloce (1)

- Un elemento essenziale dell'integrazione continua è il feedback veloce
- Le “guidelines” di eXtreme Programming parlano di un massimo di 10 minuti
- Molto spesso se c’è un problema a mantenere il build a 10 minuti, questo è provocato dai test, soprattutto quelli che fanno uso di servizi esterni, come DB

## Elemento 5: Mantenere il build veloce (2)

- In caso di build troppo lunghi, si deve fare in modo di organizzare il processo a tappe (*staged build*, o *build pipeline*)
- Si parte da un “commit build” che deve durare al massimo 10 minuti, poi si possono organizzare passaggi successivi.
- Il commit build viene usato come punto di riferimento per il ciclo di integrazione continua

## Elemento 5: Mantenere il build veloce (3)

- **Esempio: two stage build**

- Il commit build si occupa della compilazione e dell'esecuzione dei test più essenziali, utilizzando oggetti "Mock" per i test sul DB
- Una volta eseguito questo (nei 10 minuti massimi) esiste un build non affidabile al 100%, ma sufficientemente sicuro da poter permettere agli sviluppatori di basare le proprie modifiche su questo ultimo build
- La seconda fase prevede l'utilizzo di tutti i test. A questo punto può anche durare alcune ore
- Se nella seconda fase si riscontrano problemi, si cerca di creare nuovi test per la prima fase che permettano di scoprire in anticipo quanto si è trovato di problematico

## Elemento 6: clonare l'ambiente di produzione (1)

- È importante poter eseguire tutti i test in un ambiente il più possibile simile a quello di produzione
- Ogni differenza può essere motivo di errore in produzione
- Clonare significa avere le stesse versioni del software, del sistema operativo, del DB, stesse librerie, stesso HW
- Ci sono dei limiti (HW troppo caro), ma spesso i vantaggi li superano ampiamente

## Elemento 6: clonare l'ambiente di produzione (2)

- Quando l'applicazione deve poter andare in produzione su ambienti diversi (esempio: applicazione desktop) è quasi impossibile provare tutte le combinazioni
- In questi casi possono però aiutare gli ambienti virtuali (VMWare, Parallels, ecc.)

## Elemento 7: Rendere accessibile il build (1)

- Uno dei maggiori problemi nello sviluppo del software è sapere se si stanno sviluppando le funzionalità realmente desiderate dal committente
- Le persone trovano più semplice vedere qualcosa di incompleto, ma che permetta di capire meglio cosa si desidera e come esprimere
- I processi agili si basano su questo e traggono vantaggio da questo tipico comportamento umano

## **Elemento 7: Rendere accessibile il build (2)**

- In queste situazioni è importante che ogni sviluppatore abbia facile accesso all'ultimo build, o al build della sua ultima iterazione
- Lo scopo è quello di poterlo usare per dimostrazioni, test, o anche unicamente per verificare cosa è cambiato negli ultimi giorni
- Il repository deve essere organizzato in modo che ci sia una chiara sequenza storica dei build

## **Elemento 8: Visione dello stato (1)**

- CI serve anche alla comunicazione, perciò è importante che ognuno possa verificare lo stato del sistema e i cambiamenti effettuati
- I programmi di CI hanno un'interfaccia Web che permette di accedere alla storia dei cambiamenti, sapere chi ha fatto le modifiche, ecc.
- Il vantaggio di un'interfaccia Web consiste nel fatto che anche sviluppatori localizzati altrove hanno facile accesso alle informazioni

## Elemento 8: Visione dello stato (2)

- Interfaccia Web di TeamCity

The screenshot shows the TeamCity web interface. At the top, there are navigation tabs: 'Projects' (selected), 'My Changes', 'Agents (1)', and 'Build Queue (0)'. On the right, there are links for 'Administration', 'My Settings & Tools', and a search bar. Below the tabs, it says 'Welcome, sandro Logout'. A 'Configure Visible Projects' link is also present. The main area shows the 'CobraWunelli' project. It has two build configurations: 'CobraWunelli Maintenance Build Configuration' (build #563-2) and 'CobraWunelli Trunk Build Configuration' (build #680). Each configuration shows 'Tests passed: 159' and 'Tests passed: 191' respectively. The interface also indicates 'No artifacts' for both builds. At the bottom, there are buttons for 'Idle' and 'Run'.

Sandro Pedrazzini

Integrazione continua

37

## Elemento 9: Deployment automatico

- Un elemento di integrazione continua è il deployment automatico
- Questo può essere utile soprattutto se si hanno diversi deployment in ambienti diversi: automatizzare significa evitare facili fonti di errore
- Se è compresa la funzionalità di deployment in produzione, una capacità interessante è quella del rollback automatico alla versione precedente: questo permette di eliminare la “tensione” tipica del deployment

Sandro Pedrazzini

Integrazione continua

38

## Benefici della CI (1)

- Minor rischio
  - Ricordo di progetti senza integrazione continua: progetto terminato, ma incognita dell'integrazione
  - Difficile prevedere il tempo necessario di un'integrazione prevista solo alla fine del progetto
  - CI permette di sapere in ogni momento a che punto si è con il progetto e con gli errori

## Benefici della CI (2)

- Errori
  - CI non ci permette di eliminare gli errori, ma ci rende più semplice il compito di trovarli
  - Quando si introduce un bug nel sistema, se lo si scopre in fretta, diventa semplice anche toglierlo
  - Inoltre l'errore può essere solo introdotto in poche ore di lavoro dall'ultimo build stabile
  - Si eliminano i bug accumulati nel tempo

## Benefici della CI (3)

- Deployment più frequente
  - Grossa barriera che CI permette di eliminare
  - Permette di mostrare più rapidamente nuove features all'utente finale e ottenere di conseguenza un feedback più veloce
  - Attraverso feedback più veloce, utente e sviluppatore migliorano il loro rapporto di collaborazione
  - Si accorcia la distanza che esiste tipicamente tra sviluppatore e utente, decisiva per uno sviluppo software di successo

## Integrazione con le altre pratiche (1)

- L'utilizzo di CI si integra bene con altre pratiche di progettazione e sviluppo trattate
  - Test di unità
  - Utilizzo di standard nel codice
  - Refactoring
  - Cicli di sviluppo corti
  - Appartenenza comune del codice

## Integrazione con le altre pratiche (2)

- **Test di unità**

- Chi sviluppa, dovrebbe aggiungere codice di test al proprio codice (unit testing)
- Il test dovrebbe essere richiamato dopo ogni cambiamento
- Con CI il test viene richiamato automaticamente ad ogni build, quindi ad ogni modifica nel repository (regression test)

## Integrazione con le altre pratiche (3)

- **Utilizzo di standard**

- In ogni progetto si definiscono delle *guidelines* da seguire, in modo che l'intero codice segua gli stessi "standard"
- Spesso il controllo dell'aderenza allo standard è un processo manuale
- Con CI si può inserire una serie di analisi statiche del codice nello script di build, in grado di generare un report

## Integrazione con le altre pratiche (4)

- **Refactoring**

- Refactoring significa adattare il codice e la sua struttura interna senza modificare la funzionalità
- Uno degli scopi è quello di facilitare la manutenzione del codice
- CI assiste lo sviluppatore permettendo la chiamata di tool di inspection del codice all'interno del build, eseguito ad ogni modifica del repository

## Integrazione con le altre pratiche (5)

- **Cicli brevi**

- Significa che gli utenti devono avere a disposizione l'ultima versione del software funzionante il più spesso possibile
- CI si adatta bene a questa pratica, perché si integra più volte al giorno e ogni integrazione genera virtualmente un nuovo release
- Quando un sistema di integrazione continua è installato, un nuovo release viene generato con il minimo degli sforzi

## Integrazione con le altre pratiche (6)

- **Appartenenza comune (*collective ownership*)**
  - Ogni sviluppatore può lavorare a qualsiasi parte del sistema
  - Questo impedisce che ci sia un solo sviluppatore con conoscenze specifiche di un determinato argomento
  - CI aiuta questa pratica assicurando aderenza agli standard e richiamando continuamente i test di regressione

## Ridurre i rischi (1)

- **CI aiuta nel mantenere i rischi sotto controllo, permettendo di scoprire i problemi appena questi si manifestano**
- **Con CI e le pratiche correlate si riesce a creare una rete di qualità, che ci permette di fornire software di qualità più velocemente**

## Ridurre i rischi (2)

- Consideriamo i seguenti rischi
  - Software non deployable
  - Difetti scoperti tardi
  - Mancanza di visibilità del progetto
  - Software di bassa qualità
- Non si tratta di rendere attenti verso questi rischi (sono tutti noti), ma di permetterne la gestione

## Ridurre i rischi (3)

- Software non “deployabile”, impossibile creare il build

- Riesco a creare il build solo sulla mia macchina

In questo caso è importante sottolineare che il build dev'essere creato in modo indipendente dalla macchina, IDE utilizzato, configurazione specifica, ecc.  
È importante avere un server di integrazione, che usi uno script di build (ant) indipendente.

- Sincronizzazione con il DB

I test devono poter girare utilizzando l'ultima versione dello schema di DB. Lo schema di DB e i suoi dati minimi indispensabili devono trovarsi nel repository.  
I test devono essere in grado di eseguire un “drop” del DB e poi ricrearlo nuovo.

## Ridurre i rischi (4)

- **Difetti scoperti tardi**

- **Regression test**

Sappiamo che dobbiamo avere suite di test nel nostro progetto.  
Basta aggiungere la chiamata di questi test nello script di build, in questo modo verranno eseguiti dal server di integrazione ad ogni modifica.

- **Test coverage**

Esistono tool per verificare la percentuale di copertura dei test.  
Questi tool possono essere associati al processo di CI.

## Ridurre i rischi (5)

- **Mancanza di visibilità**

- **Informazioni**

È necessario che ogni sviluppatore nel progetto sia informato su ogni singola modifica effettuata al progetto.  
Al processo di CI può essere associata una notifica via email sulle modifiche effettuate o sugli eventuali problemi.

- **Visualizzazione grafica della struttura**

Se si desidera avere a disposizione l'ultima versione grafica della struttura del software (UML class diagram), lo si può fare inserendo nel sistema di CI la generazione a partire dall'ultima versione (esempio: Doxygen).

## Ridurre i rischi (6)

- **Software di bassa qualità (1)**

- **Aderenza del codice a standard predefiniti**

L'aderenza agli standard viene spesso controllata manualmente. In realtà esistono tool come Checkstyle, PMD o anche Sonar, che permettono di fare delle verifiche statiche del codice a partire da regole. Questi possono essere integrati al sistema CI.

- **Aderenza all'architettura**

Anche a livello di architettura ci possono essere delle guideline da seguire, ad esempio, il codice del data layer che non accede al codice del business layer, ecc.

Anche queste possono essere controllate da tool (JDepend, NDepend, ecc.) integrabili nel processo di CI.

## Ridurre i rischi (7)

- **Software di bassa qualità (2)**

- **Duplicazione del codice**

Codice duplicato rende più difficili le modifiche e la manutenzione del progetto.

È difficile scoprire dove abbiamo duplicazione di codice all'interno del progetto. Più il progetto è grande, più sviluppatori lavorano e maggiore è la probabilità di avere codice in più parti che esegue la stessa funzionalità.

Anche in questo caso esistono tool (utility CPD del tool di analisi metriche PMD, Simian, ecc.) che possono essere integrati nel processo di CI

## Come introdurre CI (1)

- Tutte le pratiche viste fin qui servono a trarre il massimo beneficio da CI
- Per iniziare non serve però applicarle tutte assieme
- Il primo passo è senza dubbio quello dell'automazione del processo di build
  - Mettiamo il progetto sotto il controllo di un sistema di gestione dei sorgenti (esempio: Subversion)
  - Facciamo in modo che con un unico comando si possa creare un build

## Come introdurre CI (2)

- Il passo successivo potrebbe essere quello di introdurre suite di test nel build automatico
- Se si hanno già test di unità nel sistema, la cosa è molto semplice, basta richiamarli durante il build
- Inizialmente il build verrà fatto partire a mano. Poi, si potrà richiamarlo automaticamente, con uno script, una volta al giorno per il nightly build
- Come ultimo passaggio, possiamo installare un server di integrazione con un software di CI (esempio: TeamCity)

## Qualità nei processi di sviluppo

### Integrazione continua

### Integrazione continua

- Pratica dello sviluppo software
- Ogni membro del gruppo di sviluppo integra il proprio lavoro il più frequentemente possibile, solitamente almeno una volta al giorno
- Ogni integrazione viene verificata da un *build* automatico, che lancia i test e verifica eventuali problemi di integrazione

## Integrazione continua (2)

- Molti team di sviluppo trovano che una pratica del genere serve a ridurre al minimo i problemi di integrazione
- Permette al team di sviluppare software coeso in modo molto più rapido
- Aiuta a ridurre i rischi legati allo sviluppo e all'integrazione

## Motivazione

- In molti processi software l'integrazione richiede giorni, settimane o mesi
- Questo perché l'integrazione viene eseguita come ultimo passaggio prima di andare in produzione
- Per poter andare in produzione più frequentemente bisogna trasformare l'integrazione in qualcosa di automatico (integrazione vista come “non-event”)

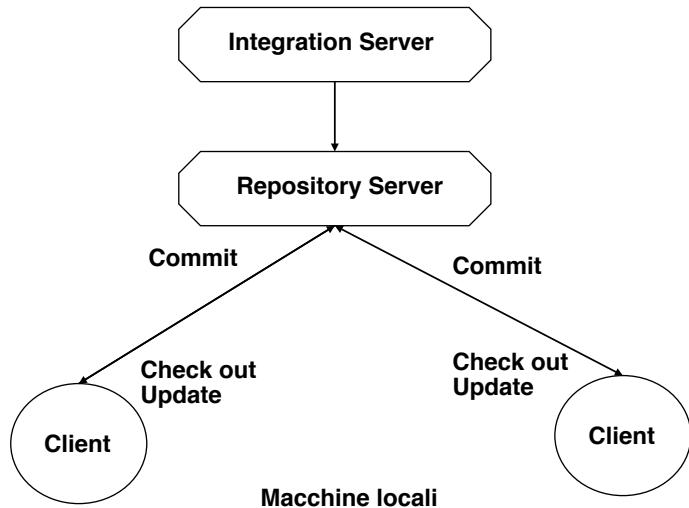
## Motivazione (2)

- Integrando regolarmente significa che ogni sviluppatore ad ogni iterazione aggiunge solo alcune ore di lavoro al progetto integrato
- Anche in questo caso (come nelle altre pratiche di XP) si tratta semplicemente di applicare in modo coerente alcune “buone abitudini”.
- Più frequentemente si integra e più l’integrazione diventa un “non-event”

## Tool

- Integrazione continua è prima di tutto una buona abitudine, perciò di base non necessitano tool particolari
- Ne esistono però alcuni (esempi: **CruiseControl**, **TeamCity**, **Jenkins**) che combinati a un sistema di gestione concorrente di sorgente (**CVS**, **Subversion**, **GIT**, ecc.) e a un sistema di build indipendente da ogni IDE (esempio: Ant, Maven, Gradle) permettono di attivare le verifiche di integrazione in modo automatico.

## Tool (2)

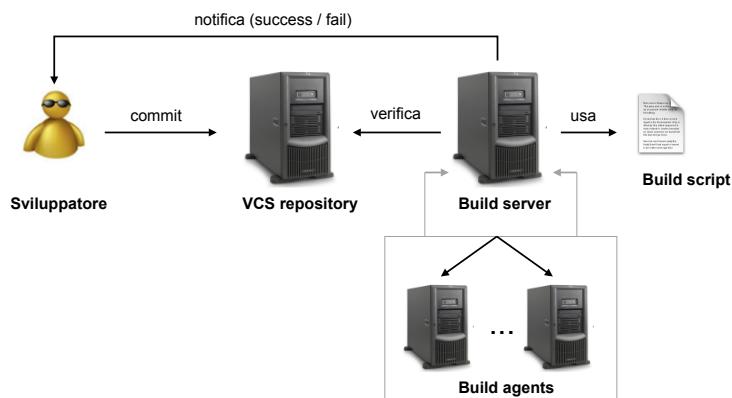


Sandro Pedrazzini

Integrazione continua

7

## Processo



Sandro Pedrazzini

Integrazione continua

8

## Tool (3)

- Esempio di build con Ant

➤ ant integrate

```
Buildfile: build.xml
clean:
all:
compile-src:
compile-tests:
integrate-db:
run-tests:
run-inspections:
delivery:
deploy:
BUILD SUCCESSFUL
Total time: 6 minutes 15 seconds
```

## Tool (4)

- Esempio di commit con Subversion (via linea di comando)

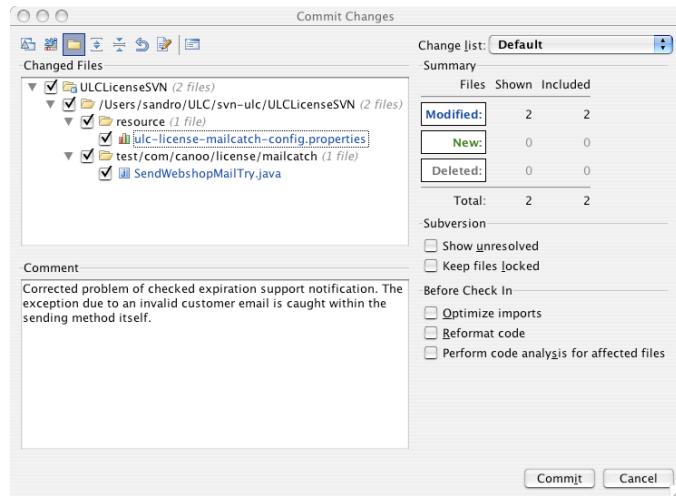
➤ svn commit -m "Aggiunta verifica della connessione"

```
Sending src/com/canoo/db/dao/DBServicesImpl.java
Transmitting file data .

Committed revision 125
```

## Tool (5)

- Esempio di commit integrato in IDE



Sandro Pedrazzini

Integrazione continua

11

## Tool (6)

- Confronto fra versioni

```

10351     mailContent.append(" no new expired support license ");
    } else {
        content.append("Expired support licenses: " + checkedSupportLicenses.size() + "\n");
        fillMailContent(mailContent, bucketsOfExpiredLicenses);
        logger.info("Send summary mail to " + baseConfig.EMAIL_ADMIN);
        MailHandler.sendSingleEmail(baseConfig.EMAIL_ADMIN, null, null, baseConfig.EMAIL_EXP_SUPPORT);
        if (baseConfig.MAIL_BE_SENT_TO_CUSTOMERS) {
            sendNotificationToSingleClients(bucketsOfExpiredLicenses);
        }
        return (SupportLicensee[]) checkedSupportLicenses.toArray(new SupportLicense[checkedSupportLicenses.size()]);
    }
}

private static void sendNotificationToSingleClients(Map expirationBuckets) throws Exception {
    if (expirationBuckets.size() == 0) {
        logger.warn("No new support license for each single client");
        Set sortedKeys = expirationBuckets.keySet();
        Iterator iter = sortedKeys.iterator();
        while (iter.hasNext()) {
            List contentList = (List) expirationBuckets.get(iter.next());
            ProductAndSupportLicensePair licensePair = (ProductAndSupportLicensePair) contentList.get(0);
            String mailCustomerText = prepareCustomerText(licensePair, contentList);
            MailHandler.sendSingleEmail(licensePair.getClientEmail(), prepareSellerAddress(licensePair));
        }
    }
}

private static String prepareSellerAddress(String secondContactEmail) {
    if (baseConfig.RESELLER_AB_CO) {
        if (secondContactEmail != null && secondContactEmail.equals("")) {
            return secondContactEmail;
        }
    }
    return null;
}

private static String prepareCustomerText(ProductAndSupportLicensePair licensePair, List contentList) {
    StringBuffer text = new StringBuffer();
    text.append("Dear ").append(licensePair.getFirstName()).append(" ").append(licensePair.getLastName());
    contentList.iterator();
    Iterator iter = contentList.iterator();
    while (iter.hasNext()) {
        ProductAndSupportLicensePair pair = (ProductAndSupportLicensePair) iter.next();
        text.append(pair.getCompanyName());
        text.append(" (").append(pair.getLicensee().getExtendedNumber()).append(")\n");
    }
    text.append(baseConfig.CUSTOMER_MAIL_TEXT);
    return text.toString();
}

private void fillBucketOfRelatedExpirationLicenses(Map expirationBucket, SupportLicensee supp
ProductLicensee) {
    RelatedProductLicenses relatedProductLicenses = dataservice.loadLicenses(supportLicense);
    for (int i = 0; i < relatedProductLicenses.size(); i++) {
        Client client = relatedProductLicenses[i].getClient();
        String key = client.getEmail() + client.getCompany() + supportLicense.getExpirationDate();
        if (client2 != null) {
            key = key + client2.getEmail() + client2.getCompany(); // singleLicense.getsize();
        }
        List contentList = (List) expirationBucket.get(key);
        if (contentList == null) {
    
```

10351 mailContent.append(" no new expired support license ");
 } else {
 content.append("Expired support licenses: " + checkedSupportLicenses.size() + "\n");
 fillMailContent(mailContent, bucketsOfExpiredLicenses);
 logger.info("Send summary mail to " + baseConfig.EMAIL\_ADMIN);
 MailHandler.sendSingleEmail(baseConfig.EMAIL\_ADMIN, null, null, baseConfig.EMAIL\_EXP\_SUPPORT);
 if (baseConfig.MAIL\_BE\_SENT\_TO\_CUSTOMERS) {
 sendNotificationToSingleClients(bucketsOfExpiredLicenses);
 }
 return (SupportLicensee[]) checkedSupportLicenses.toArray(new SupportLicense[checkedSupportLicenses.size()]);
 }
}

private static void sendNotificationToSingleClients(Map expirationBuckets) throws Exception {
 if (expirationBuckets.size() == 0) {
 logger.warn("No new support license for each single client");
 Set sortedKeys = expirationBuckets.keySet();
 Iterator iter = sortedKeys.iterator();
 while (iter.hasNext()) {
 List contentList = (List) expirationBuckets.get(iter.next());
 ProductAndSupportLicensePair licensePair = (ProductAndSupportLicensePair) contentList.get(0);
 String mailCustomerText = prepareCustomerText(licensePair, contentList);
 MailHandler.sendSingleEmail(licensePair.getClientEmail(), prepareSellerAddress(licensePair));
 }
 }
}

private static String prepareSellerAddress(String secondContactEmail) {
 if (baseConfig.RESELLER\_AB\_CO) {
 if (secondContactEmail != null && secondContactEmail.equals("")) {
 return secondContactEmail;
 }
 }
 return null;
}

private static String prepareCustomerText(ProductAndSupportLicensePair licensePair, List contentList) {
 StringBuffer text = new StringBuffer();
 contentList.iterator();
 Iterator iter = contentList.iterator();
 while (iter.hasNext()) {
 ProductAndSupportLicensePair pair = (ProductAndSupportLicensePair) iter.next();
 text.append("Dear ").append(pair.getFirstName()).append(" ").append(pair.getLastName());
 iter.next();
 StringWriter stringWriter = new StringWriter();
 content.append(message).append("\n");
 content.append(exception).append("\n");
 e.printStackTrace(stringWriter);
 MailHandler.sendSingleEmail(baseConfig.EMAIL\_ADMIN, null, null, "Problem in ulc/che
 

6 differences Deleted Changed Inserted

Sandro Pedrazzini

Integrazione continua

12

## Tool (8)

- Mail ricevuta da Cruise Control dopo il build

### BUILD COMPLETE - build.180

Date of build: 01/07/2008 16:21:50  
Time to build: 10 seconds  
Last changed: 01/07/2008 16:16:51  
Last log entry: Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Modifications since last successful build: (6)

modified sandro /ULCLicense/trunk/src/com/canoo/license/base/data/transaction/TransactionContext.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
added sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification-3.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/src/com/canoo/license/base/support/CheckExpiringSupport.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/build.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicenseTemplates/trunk/ULCBase/ulc-support-additions.sql	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Sandro Pedrazzini

Integrazione continua

13

## Tool (9)

### Progetti in TeamCity

Welcome, sandro Logout

Projects My Changes Agents (1) Build Queue (0) Administration My Settings & Tools Configure Visible Projects

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- **CobraWunelli Maintenance Build Configuration** Idle Run
- #build.563-2 Tests passed: 159
- No artifacts Changes (1) 10 Feb 17:56 (7m:37s)
- **CobraWunelli Trunk Build Configuration** Idle Pending (1) Run
- #build.680 Tests passed: 191
- No artifacts Changes (1) 25 Feb 15:33 (12m:09s)

Sandro Pedrazzini

Integrazione continua

14

## Tool (10)

- History di TeamCity

The screenshot shows the TeamCity web interface for the project 'CobraWunelli'. The top navigation bar includes 'Projects', 'My Changes', 'Agents (1)', 'Build Queue (0)', 'Administration', 'My Settings & Tools', and a search bar. The main content area displays the 'History' tab of the build configuration 'CobraWunelli Trunk Build Configuration'. It shows a table of recent builds with columns for '#', 'Results', 'Artifacts', 'Changes', 'Started', 'Duration', 'Agent', 'Tags', and a 'Pin' icon. The table lists builds from #build.680 to #build.666. Build #680 is green (Tests passed: 191). Builds #678, #677, #676, #672, #671, #670, and #669 are green (Tests passed: 185, 172, 172, 175, 175, 175, 175). Build #668 is red (Tests failed: 1 (1 new), passed: 171). Build #666 is red ([Execution timeout] Tests passed: 171). The status bar at the bottom indicates 'Integrazione continua' and the page number '15'.

## Esempio (1)

- Supponiamo di voler aggiungere una nuova funzionalità ad un programma esistente

- Come primo passaggio devo scaricare la versione più aggiornata sulla mia macchina locale (check-out o update, nel caso avessi già una versione locale non aggiornata)
- Quando ho la versione sulla mia macchina, posso iniziare a fare le aggiunte desiderate
- Questo significa non solo aggiungere funzionalità, ma anche adattare vecchio codice, aggiungere e adattare test

## Esempio (2)

- Appena terminato con la nuova funzionalità e i vari test, posso creare il build sulla mia macchina locale
- Questo significa ricompilare tutto, creare e includere le varie librerie, eseguire i test
- Posso considerare di aver terminato il lavoro sulla macchina locale solo quando tutto funziona senza errori
- Ora che il build locale funziona, posso pensare di eseguire un “commit” sul repository comune

## Esempio (3)

- Non ho ancora finito: a questo punto si tratta di eseguire il build sulla macchina di integrazione
- Solo se anche questo build ha successo, si può affermare che i cambiamenti eseguiti fanno parte dell’applicazione
- Il build di integrazione può essere eseguito manualmente oppure automaticamente con tool come TeamCity, che, eseguendo un polling continuo sul repository per determinare se ci sono stati aggiornamenti, fa partire il build quando necessario

## Conflitti (1)

- Se ci sono conflitti a causa di due aggiornamenti che si accavallano, solitamente la cosa viene notata dal secondo che esegue commit
- Significa che dal suo ultimo update, qualcun altro ha eseguito un commit
- Il caso viene risolto localmente con un nuovo update (ed eventuali adattamenti) prima del commit

## Conflitti (2)

- Se il conflitto non si manifesta durante il commit (perché non sono stati toccati gli stessi file), ma esiste comunque una inconsistenza, questa viene scoperta durante il build di integrazione
- In entrambi i casi il problema viene scoperto velocemente
- La cosa più urgente da fare in questi casi è riparare il build

## Conflitti (3)

- In un ambiente di integrazione continua non si dovrebbe mai lasciare un “failed build” troppo a lungo
- Un buon team dovrebbe avere più di un build corretto al giorno
- Ognuno può quindi sviluppare a partire dall’ultima versione del build, mantenendo quindi il delta tra sviluppo e successiva integrazione il più piccolo possibile

## Elementi di CI

- Quanto visto nell’esempio precedente dimostra CI (continuous integration) nel lavoro di tutti i giorni
- Vediamo quali sono gli elementi essenziali affinché tutto questo possa avvenire in modo naturale e senza grossi problemi

## Elemento 1: Un solo repository

- I progetto software richiedono la gestione di parecchi file: utilizzare un sistema di versioning control
- Fare in modo che sia accessibile a tutti, anche da remoto
- Nel repository dovrebbe esserci tutto quanto server per eseguire il build, inclusi script, property file, librerie, ecc.
- Regola: dev'essere possibile eseguire un check out su una macchina "verGINE" ed poter subito eseguire un build

## Elemento 2: Build automatico (1)

- Build significa trasformare i sorgenti in un sistema funzionante
- Questo può essere un processo complicato che include compilazione, spostamento di file, caricamento di uno schema di DB, ecc.
- Tutto questo può essere automatizzato ed è buona cosa che lo sia, perché fa risparmiare parecchio tempo

## Elemento 2: Build automatico (2)

- Ambienti di build automatico ne esistono parecchi: Make in Unix, Ant, Maven, Gradle nel mondo Java, Nant o MSBuild per .NET, ecc.
- A dipendenza delle necessità si deve poter creare build in modo condizionale, avendo a disposizione diversi target (build con codice di test integrato, con verifiche diverse, ecc.)
- Il build può essere creato via IDE, ma dev'essere in ogni caso possibile creare un master sul server, senza IDE

## Elemento 3: Build self-testing

- Build non significa solo compilazione e link, un programma compilato correttamente può avere errori di esecuzione
- Il modo migliore per verificare il funzionamento è inserire la chiamata ai test nel processo di build
- Se un test non passa, il build deve fallire

## **Elemento 4: Si integra ogni giorno (1)**

- L'integrazione è anche un mezzo per comunicare agli altri sviluppatori quali modifiche abbiamo apportato al codice
- Integrando regolarmente si scoprono prima eventuali conflitti di sincronizzazione tra sviluppatori e si possono correggere velocemente
- Conflitti che rimangono irrisolti per giorni o settimane, sono difficili da riparare

## **Elemento 4: Si integra ogni giorno (2)**

- Regola: ogni sviluppatore dovrebbe eseguire un commit almeno una volta al giorno
- Commit frequenti (anche più volte al giorno) incoraggiano lo sviluppatore a suddividere il suo lavoro in fasi di alcune ore l'una. Questo permette di “sentire” il progredire del progetto

## Elemento 5: Mantenere il build veloce (1)

- Un elemento essenziale dell'integrazione continua è il feedback veloce
- Le “guidelines” di eXtreme Programming parlano di un massimo di 10 minuti
- Molto spesso se c’è un problema a mantenere il build a 10 minuti, questo è provocato dai test, soprattutto quelli che fanno uso di servizi esterni, come DB

## Elemento 5: Mantenere il build veloce (2)

- In caso di build troppo lunghi, si deve fare in modo di organizzare il processo a tappe (*staged build*, o *build pipeline*)
- Si parte da un “commit build” che deve durare al massimo 10 minuti, poi si possono organizzare passaggi successivi.
- Il commit build viene usato come punto di riferimento per il ciclo di integrazione continua

## Elemento 5: Mantenere il build veloce (3)

- **Esempio: two stage build**

- Il commit build si occupa della compilazione e dell'esecuzione dei test più essenziali, utilizzando oggetti "Mock" per i test sul DB
- Una volta eseguito questo (nei 10 minuti massimi) esiste un build non affidabile al 100%, ma sufficientemente sicuro da poter permettere agli sviluppatori di basare le proprie modifiche su questo ultimo build
- La seconda fase prevede l'utilizzo di tutti i test. A questo punto può anche durare alcune ore
- Se nella seconda fase si riscontrano problemi, si cerca di creare nuovi test per la prima fase che permettano di scoprire in anticipo quanto si è trovato di problematico

## Elemento 6: clonare l'ambiente di produzione (1)

- È importante poter eseguire tutti i test in un ambiente il più possibile simile a quello di produzione
- Ogni differenza può essere motivo di errore in produzione
- Clonare significa avere le stesse versioni del software, del sistema operativo, del DB, stesse librerie, stesso HW
- Ci sono dei limiti (HW troppo caro), ma spesso i vantaggi li superano ampiamente

## **Elemento 6: clonare l'ambiente di produzione (2)**

- Quando l'applicazione deve poter andare in produzione su ambienti diversi (esempio: applicazione desktop) è quasi impossibile provare tutte le combinazioni
- In questi casi possono però aiutare gli ambienti virtuali (VMWare, Parallels, ecc.)

## **Elemento 7: Rendere accessibile il build (1)**

- Uno dei maggiori problemi nello sviluppo del software è sapere se si stanno sviluppando le funzionalità realmente desiderate dal committente
- Le persone trovano più semplice vedere qualcosa di incompleto, ma che permetta di capire meglio cosa si desidera e come esprimere
- I processi agili si basano su questo e traggono vantaggio da questo tipico comportamento umano

## **Elemento 7: Rendere accessibile il build (2)**

- In queste situazioni è importante che ogni sviluppatore abbia facile accesso all'ultimo build, o al build della sua ultima iterazione
- Lo scopo è quello di poterlo usare per dimostrazioni, test, o anche unicamente per verificare cosa è cambiato negli ultimi giorni
- Il repository deve essere organizzato in modo che ci sia una chiara sequenza storica dei build

## **Elemento 8: Visione dello stato (1)**

- CI serve anche alla comunicazione, perciò è importante che ognuno possa verificare lo stato del sistema e i cambiamenti effettuati
- I programmi di CI hanno un'interfaccia Web che permette di accedere alla storia dei cambiamenti, sapere chi ha fatto le modifiche, ecc.
- Il vantaggio di un'interfaccia Web consiste nel fatto che anche sviluppatori localizzati altrove hanno facile accesso alle informazioni

## Elemento 8: Visione dello stato (2)

- Interfaccia Web di TeamCity

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- = **CobraWunelli Maintenance Build Configuration**  
#build.563-2 Tests passed: 159 | No artifacts Changes (1) 10 Feb 17:56 (7m:37s)  
Idle Run
- = **CobraWunelli Trunk Build Configuration**  
#build.680 Tests passed: 191 | No artifacts Changes (1) 25 Feb 15:33 (12m:09s)  
Idle Pending (1) Run

Sandro Pedrazzini Integrazione continua 37

## Elemento 9: Deployment automatico

- Un elemento di integrazione continua è il deployment automatico
- Questo può essere utile soprattutto se si hanno diversi deployment in ambienti diversi: automatizzare significa evitare facili fonti di errore
- Se è compresa la funzionalità di deployment in produzione, una capacità interessante è quella del rollback automatico alla versione precedente: questo permette di eliminare la “tensione” tipica del deployment

## Benefici della CI (1)

- Minor rischio
  - Ricordo di progetti senza integrazione continua: progetto terminato, ma incognita dell'integrazione
  - Difficile prevedere il tempo necessario di un'integrazione prevista solo alla fine del progetto
  - CI permette di sapere in ogni momento a che punto si è con il progetto e con gli errori

## Benefici della CI (2)

- Errori
  - CI non ci permette di eliminare gli errori, ma ci rende più semplice il compito di trovarli
  - Quando si introduce un bug nel sistema, se lo si scopre in fretta, diventa semplice anche toglierlo
  - Inoltre l'errore può essere solo introdotto in poche ore di lavoro dall'ultimo build stabile
  - Si eliminano i bug accumulati nel tempo

## Benefici della CI (3)

- Deployment più frequente
  - Grossa barriera che CI permette di eliminare
  - Permette di mostrare più rapidamente nuove features all'utente finale e ottenere di conseguenza un feedback più veloce
  - Attraverso feedback più veloce, utente e sviluppatore migliorano il loro rapporto di collaborazione
  - Si accorcia la distanza che esiste tipicamente tra sviluppatore e utente, decisiva per uno sviluppo software di successo

## Integrazione con le altre pratiche (1)

- L'utilizzo di CI si integra bene con altre pratiche di progettazione e sviluppo trattate
  - Test di unità
  - Utilizzo di standard nel codice
  - Refactoring
  - Cicli di sviluppo corti
  - Appartenenza comune del codice

## Integrazione con le altre pratiche (2)

- **Test di unità**

- Chi sviluppa, dovrebbe aggiungere codice di test al proprio codice (unit testing)
- Il test dovrebbe essere richiamato dopo ogni cambiamento
- Con CI il test viene richiamato automaticamente ad ogni build, quindi ad ogni modifica nel repository (regression test)

## Integrazione con le altre pratiche (3)

- **Utilizzo di standard**

- In ogni progetto si definiscono delle *guidelines* da seguire, in modo che l'intero codice segua gli stessi "standard"
- Spesso il controllo dell'aderenza allo standard è un processo manuale
- Con CI si può inserire una serie di analisi statiche del codice nello script di build, in grado di generare un report

## Integrazione con le altre pratiche (4)

- **Refactoring**

- Refactoring significa adattare il codice e la sua struttura interna senza modificare la funzionalità
- Uno degli scopi è quello di facilitare la manutenzione del codice
- CI assiste lo sviluppatore permettendo la chiamata di tool di inspection del codice all'interno del build, eseguito ad ogni modifica del repository

## Integrazione con le altre pratiche (5)

- **Cicli brevi**

- Significa che gli utenti devono avere a disposizione l'ultima versione del software funzionante il più spesso possibile
- CI si adatta bene a questa pratica, perché si integra più volte al giorno e ogni integrazione genera virtualmente un nuovo release
- Quando un sistema di integrazione continua è installato, un nuovo release viene generato con il minimo degli sforzi

## Integrazione con le altre pratiche (6)

- **Appartenenza comune (*collective ownership*)**
  - Ogni sviluppatore può lavorare a qualsiasi parte del sistema
  - Questo impedisce che ci sia un solo sviluppatore con conoscenze specifiche di un determinato argomento
  - CI aiuta questa pratica assicurando aderenza agli standard e richiamando continuamente i test di regressione

## Ridurre i rischi (1)

- **CI aiuta nel mantenere i rischi sotto controllo, permettendo di scoprire i problemi appena questi si manifestano**
- **Con CI e le pratiche correlate si riesce a creare una rete di qualità, che ci permette di fornire software di qualità più velocemente**

## Ridurre i rischi (2)

- Consideriamo i seguenti rischi
  - Software non deployable
  - Difetti scoperti tardi
  - Mancanza di visibilità del progetto
  - Software di bassa qualità
- Non si tratta di rendere attenti verso questi rischi (sono tutti noti), ma di permetterne la gestione

## Ridurre i rischi (3)

- Software non “deployabile”, impossibile creare il build

- Riesco a creare il build solo sulla mia macchina

In questo caso è importante sottolineare che il build dev'essere creato in modo indipendente dalla macchina, IDE utilizzato, configurazione specifica, ecc.  
È importante avere un server di integrazione, che usi uno script di build (ant) indipendente.

- Sincronizzazione con il DB

I test devono poter girare utilizzando l'ultima versione dello schema di DB. Lo schema di DB e i suoi dati minimi indispensabili devono trovarsi nel repository.  
I test devono essere in grado di eseguire un “drop” del DB e poi ricrearlo nuovo.

## Ridurre i rischi (4)

- **Difetti scoperti tardi**

- **Regression test**

Sappiamo che dobbiamo avere suite di test nel nostro progetto.  
Basta aggiungere la chiamata di questi test nello script di build, in questo modo verranno eseguiti dal server di integrazione ad ogni modifica.

- **Test coverage**

Esistono tool per verificare la percentuale di copertura dei test.  
Questi tool possono essere associati al processo di CI.

## Ridurre i rischi (5)

- **Mancanza di visibilità**

- **Informazioni**

È necessario che ogni sviluppatore nel progetto sia informato su ogni singola modifica effettuata al progetto.  
Al processo di CI può essere associata una notifica via email sulle modifiche effettuate o sugli eventuali problemi.

- **Visualizzazione grafica della struttura**

Se si desidera avere a disposizione l'ultima versione grafica della struttura del software (UML class diagram), lo si può fare inserendo nel sistema di CI la generazione a partire dall'ultima versione (esempio: Doxygen).

## Ridurre i rischi (6)

- **Software di bassa qualità (1)**

- **Aderenza del codice a standard predefiniti**

L'aderenza agli standard viene spesso controllata manualmente. In realtà esistono tool come Checkstyle, PMD o anche Sonar, che permettono di fare delle verifiche statiche del codice a partire da regole. Questi possono essere integrati al sistema CI.

- **Aderenza all'architettura**

Anche a livello di architettura ci possono essere delle guideline da seguire, ad esempio, il codice del data layer che non accede al codice del business layer, ecc.

Anche queste possono essere controllate da tool (JDepend, NDepend, ecc.) integrabili nel processo di CI.

## Ridurre i rischi (7)

- **Software di bassa qualità (2)**

- **Duplicazione del codice**

Codice duplicato rende più difficili le modifiche e la manutenzione del progetto.

È difficile scoprire dove abbiamo duplicazione di codice all'interno del progetto. Più il progetto è grande, più sviluppatori lavorano e maggiore è la probabilità di avere codice in più parti che esegue la stessa funzionalità.

Anche in questo caso esistono tool (utility CPD del tool di analisi metriche PMD, Simian, ecc.) che possono essere integrati nel processo di CI

## Come introdurre CI (1)

- Tutte le pratiche viste fin qui servono a trarre il massimo beneficio da CI
- Per iniziare non serve però applicarle tutte assieme
- Il primo passo è senza dubbio quello dell'automazione del processo di build
  - Mettiamo il progetto sotto il controllo di un sistema di gestione dei sorgenti (esempio: Subversion)
  - Facciamo in modo che con un unico comando si possa creare un build

## Come introdurre CI (2)

- Il passo successivo potrebbe essere quello di introdurre suite di test nel build automatico
- Se si hanno già test di unità nel sistema, la cosa è molto semplice, basta richiamarli durante il build
- Inizialmente il build verrà fatto partire a mano. Poi, si potrà richiamarlo automaticamente, con uno script, una volta al giorno per il nightly build
- Come ultimo passaggio, possiamo installare un server di integrazione con un software di CI (esempio: TeamCity)

# Software Engineering: DP and Refactorings

## 1. Structure

Look for information in Internet on the *Composite* pattern, study it, and use it to structure the hierarchy of organizational units within a big company.

Each organizational unit (department, division, section, office, etc.) has its own associated persons and can contain further subordinated organizational units, each own, again, with specific associated persons and further subunits.

In the *Composite* pattern implementation, all persons will be modeled as leaf component, whereas each organizational unit should be modeled as composite.

Once created the structure, you want to print, in the simplest way, the names of all persons associated to a certain unit, including the names of the persons of the subordinated units.

```
Component researchDivision = new ResearchDivision(...);  
...  
researchDivision.printAll();
```

## 2. Functionality

Look for information in Internet on the *Visitor* pattern, study it, and use it in the preceding program to independently implement more types of visualizations of the organizational structure.

Start replicating the “printAll()” functionality realized in the preceding program.

Analyze and discuss the advantages of using the *Visitor* approach, comparing it with the alternative of implementing each new kind of visualizations within the *Composite* structure.

To verify the correct *Visitor* approach, use it also for a different kind of functionality, as for example, delivering the number of persons within a given unit.

## Refactoring

### Introduction and Examples

## Content

- What is refactoring
- Why refactoring?
- Bad Code
- Software entropy
- Some examples

## Refactoring (1)

### Definitions

- Modify an existing software so that the external functionalities remain unchanged, but the internal structure is improved.  
Improved to further changes and improved for clarity.
- Disciplined way to improve and "clean up" the code, reducing to a minimum the probability of introducing errors.

## Refactoring (2)

### “Commonplace”:

First you need a complete design, then the code.

Over time the code is changed. The integrity of the initial design fails, the internal structure degrades.

It switches from "SW engineering" to just "coding".

## Refactoring (3)

### Better:

You need to work on an initial design. It is, however, during the implementation of system that you learn how to improve the design.

Having the ability to perform refactoring means being able to adapt and continually recreate the design during implementation.

Work with very short cycles.

## Why refactoring? (1)

### With “Refactoring” you improve the software design

Without refactoring the design of a software tends to decay.

Many changes tend to be changes for short-term purposes, without adequate understanding of the design, so the code loses structure and becomes unreadable in a short time.

You enter into a vicious circle.

A regular refactoring exercise helps maintaining a well structured code.

## Why refactoring? (2)

### With “Refactoring” you improve the code readability

Programming is communicating with the computer.  
We often forget, however, that the computer is not the only one having to read and interpret the code.

More people will have to read and understand it in the future.

And it often happens that these are the same people who wrote the software...

## Why refactoring? (3)

### With “Refactoring” you improve your ability to maintain the software

What does it matter if the compiler needs two steps longer to compile the code, if the change we did improves the software readability and reduces the comprehension time of hours or days to the person who will read it?

It's normal to think only at the computer during a first approach to programming.

But at a later stage you must stop and run a few cycles of refactoring.

## Why refactoring? (4)

### With “Refactoring” you discover errors in the code

Performing refactoring increases your code understanding, you simplify structures and you find bugs, often in advance in respect to their runtime appearance.

*I am not a great programmer,  
I am just a good programmer with great habits*

Kent Beck

## Why refactoring? (5)

### With “Refactoring” you program faster

It is a consequence of the previous points.

You would think the opposite.

It is clear that there is an improvement in the quality: improving design, readability and correcting the errors improve it.

**However, is there a slow down due to these activities?**

**No, because they are used to maintain the development speed high .**

A design that degenerates over time slows down exponentially the development and increases the likelihood of entering new bugs at each change.

## Bad Code

- We know good code matters, because we have had to deal for so long with its lack.
- There are companies gone out of business because of code got worse and worse until you could not manage it any longer.
- If you are a programmer of any experience then you have felt many times the impediment of bad code.

## Bad Code (2)

- We have all looked at the mess we have just made and then have chosen to leave it for another day.
- We have felt the relief of seeing our messy program work and deciding that a working mess is better than nothing.
- We have all said we'd go back and clean it up later...

## Le Blanc's law

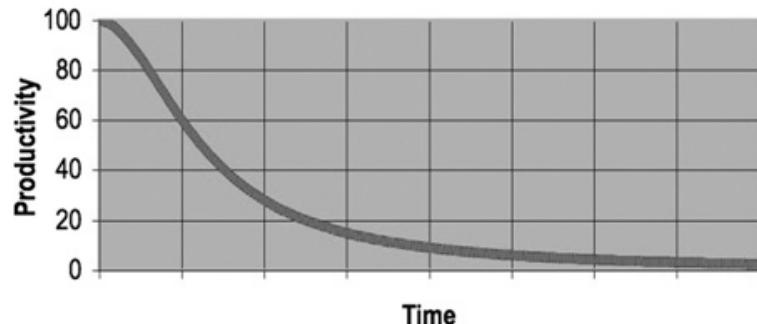
- There are several popular variants of "I'll clean it up later":
  - I'll fix that bug later.
  - I'll verify with the customer that I've built what they actually need later.
  - I'll write unit tests later.
  - I'll remove the fragility from the unit tests later.
  - I'll make the unit tests readable later.
  - I'll remove that copy/paste duplication later.
  - I'll remove that workaround/hot fix/complete hack later.
- The problem is, we usually don't get around to doing any of those things we plan to do "*later*".
- Le Blanc's law: "Later equals Never."

## The Cost of a Mess

- Everyone has been slowed down by someone else's or by his own's messy code
- The degree of the slow down can be significant
- Every addition or modification to the system requires that the tangles and knots in the messy code be understood, so that more tangles and knots can be added...

## The Cost of a Mess (2)

- Over time the mess becomes so big and so deep, that you can not clean it up.



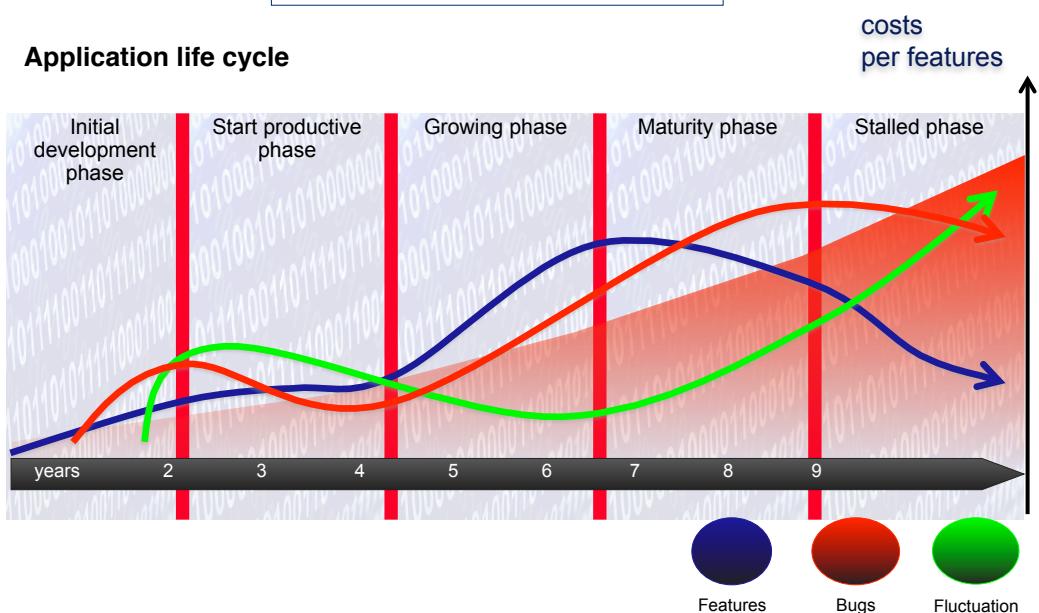
Sandro Pedrazzini

Refactoring

15

## The Cost of a Mess (3)

### Application life cycle



Sandro Pedrazzini

Refactoring

16

## Professional Survival

- At a certain point the team informs the management that they cannot continue to develop in this way. They demand a redesign.
- Management does not want to extend the resources on a whole new redesign, but they cannot deny that the productivity is low.
- A new system will be built, but the current one must be maintained.

## Professional Survival (2)

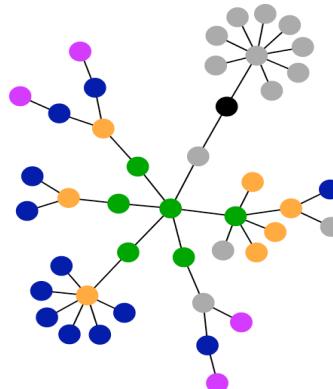
- Now there are two teams in race. The team in charge of the new system must build a new one that does everything that the old one does.  
Not only that: they have to keep up with the changes that are continuously being made to the old system.
- Management will not replace the old system until the new one can do everything.
- This race can go on for a long time.

## Professional Survival (3)

- If nothing has changed in the attitude, when the new system is done the result will be:
  - The original members of the team are long gone
  - The current members are demanding that the new system be redesigned because it is a mess
- **Conclusion:**  
spending time keeping your code clean is not just cost effective: it is a matter of professional survival

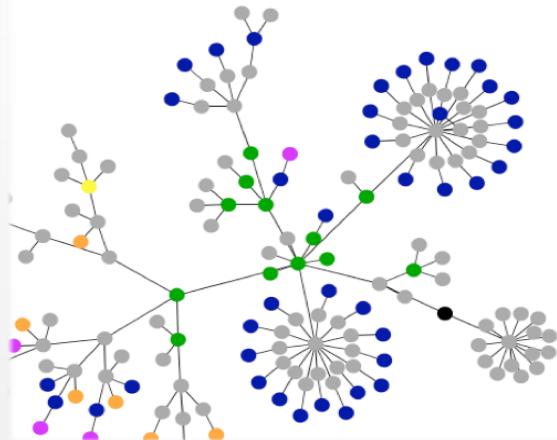
## Software entropy (1)

Our application today...



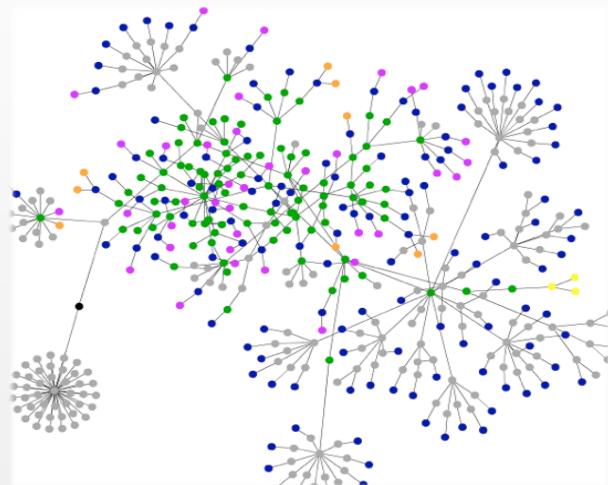
## Software entropy (2)

Our application in 5 years...



## Software entropy (3)

Our application in 10 years...

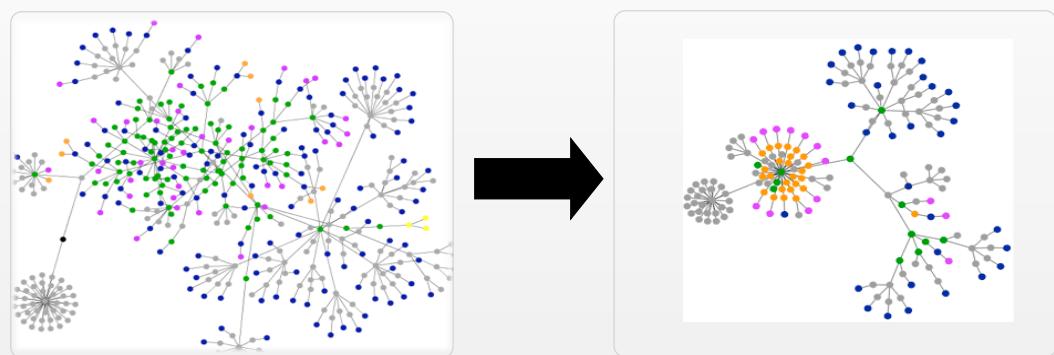


## Software entropy (4)

- A used software program always needs changes
- Each time you modify a program, its complexity grows, unless you actively keep the growth under control

## Software entropy (6)

Refactoring !



## Fixing broken windows

Through step by step code refactoring, you reduce the software entropy.

**“Fixing Broken Windows:  
Restoring Order and Reducing Crime in Our Communities”**

*If you do not repair the first broken window, there will be other broken ones in short times and the deterioration is faster*

A. Hunt & D. Thomas

## Technical debt

- **Result of software entropy**
- **The rush in developing new functionalities leads to forget to include the "ordinary" maintenance in the cost estimates**
- **If you are not used to continuously clean your code you will incur into the "technical debt"**

## Technical debt (2)

- Why do we call it “technical debt”? Because at some point you will need to return it
- The longer you wait, the greater the accumulated interest...
- Not only interest: the higher the debt, the higher the chance to add another one in future

## Technical Debt (3)

*On a large, long term project / product, Scrum (done poorly) creates technical debt.*

*Too much technical debt left unpaid results in a Legacy System.*

Clinton Keith, The Art and Business of Making Games

## Refactoring 1: replace constructor

**Title:**

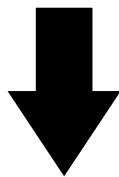
Replace Constructor with Factory Method

**Target:**

Isolate the creation of objects so that this is completely under control and it is possible to add operations to the simple creation.

## Scheme

```
public Employee(int type){  
    this.type = type;  
}
```



```
public static Employee create(int type){  
    return new Employee(type);  
}
```

## Motivations

- A typical case is when you need to replace classification code with code using subclasses

### => Replace type Code with Strategy/State

- You know where the object has to be constructed, but you want to isolate the choice of the concrete element.
- Need to add an operation call during the object's construction.
- Need to catch an exception during creation of the object.

## Procedure

1. Create a factory method, inserting a call to the current constructor
2. Replace all constructor's calls in the code with calls to the factory method

### Compile and test

3. Declare the constructor as *private*.

### Compile and test

**Example (1)**

```
public class Employee {  
    private int type;  
  
    static final int DEVELOPER = 0;  
    static final int VENDOR = 1;  
    static final int SYSADMIN= 2;  
  
    public Employee(int type){  
        this.type = type;  
    }  
}
```

1. Create a factory method

```
static Employee create(int type){  
    return new Employee(type);  
}
```

**Example (2)**

2. Replace all constructor's calls in the code with calls to the factory method

```
...  
Employee emp = Employee.create(Employee.DEVELOPER);  
...
```

**Compile and test**

3. Declare the construction as *private*.

```
public class Employee
{
    ...
    private Employee (int type){
        this.type = type;
    }
    ...
}
```

### Compile and test

So far we do not see a big advantage.

The fact that we isolated the constructor call, however, has positive implications if we replace the type code with polymorphism.

This way we can hide the creation of different subclasses in the *create()* method:

```
public static Employee create(int type){
    switch(type){
        case DEVELOPER:
            return new Developer();
        case VENDOR:
            return new Vendor();
        case SYSADMIN:
            return new SysAdmin();
        default:
            throw new IllegalArgumentException("...");
    }
}
```

Again we have the “switch problem”

Properly using the polymorphism, however, we can avoid it, except just in the factory method.

One way to fully eliminate it, is using Reflection

### Using Reflection

#### Constructor:

```
public static Employee create(String name){  
    try{  
        return (Employee)Class.forName(name).newInstance();  
    }catch(Exception e){  
        throw new IllegalArgumentException("...");  
    }  
}
```

#### Call:

```
Employee emp = Employee.create("Developer");
```

## Example (7)

### Reflection

Advantages:

No need to change the *create()* method in the case of addition of a new subclass.

The choice of the class to be instantiated can be specified in a configuration file.

Disadvantages:

No check at compile time (in the case of the configuration file, however, some IDEs provide a support and an adequate control)

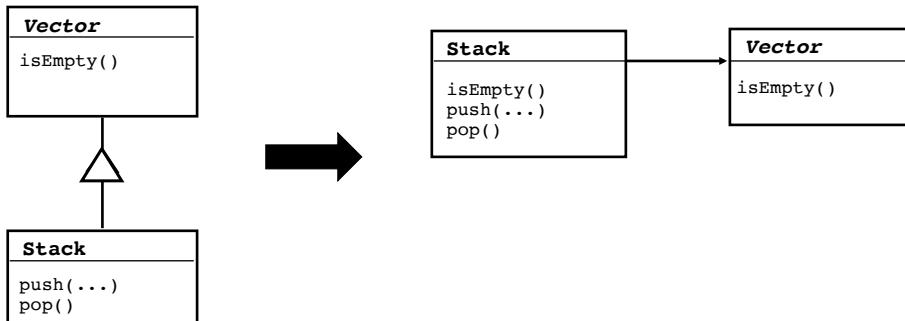
## Refactoring 2: replace inheritance

### Title:

Replace Inheritance with Delegation

### Target:

Replace an inheritance relationship between two classes by a composition relationship.



- The inheritance is useful and can be quickly applied to reuse code. It happens, however, to discover that it is NOT the best solution.
- It can happen to inherit for convenience, but then you realize that most of the functionality of the superclass is not needed. You end up with a wrong interface on your class.
- The superclass uses and loads a lot of data, not needed by the subclass, slowing down the creation of the latter (with delegation you can manage a *lazy loading* approach)
- You can definitely live with such situations, but you get a code that does not reflect the real situation of the program

## Motivations (2)

- Using the composition (delegation) it is clear that you only need and use part of the code of the delegate object.
- You have full control over the interface aspects and you can decide what to keep and what to ignore from it.
- The added cost consists on the delegation methods, which are, however, very simple.

## Procedure

1. Create a field in the subclass that refers to an instance of the superclass.  
Initialize it with “this”.
2. Change every method defined in the subclass so that it uses the delegate field.

### Compile and test

3. Remove the specification of subclass and replace the initialization of the delegate field with a new instance of a superclass object .
4. Add a method for each method of the superclass used directly by the client application.

### Compile and test

## Example (1)

A good example of inappropriate use of inheritance is the Java class *Stack* class, inheriting from *Vector*

```
public class Stack<E> extends Vector<E> {

    public void push(E element){
        insertElementAt(element,0);
    }

    public E pop(){
        E result = firstElement();
        removeElementAt(0);
        return result;
    }
}
```

## Example (2)

### Observations

Looking at the functionalities generally required by a client application, you realize that the required actions for *Stack* are basically four:

- *push()*
- *pop()*
- *size()*
- *isEmpty()*

Only two *Vector* operations are needed (*size()*, *isEmpty()*)

### Example (3)

1. Create a field in the subclass that refers to an instance of the superclass.  
Initialize it with “this”.

```
private Vector vector = this;
```

2. Change every method defined in the subclass so that it uses the delegate field.

```
public void push(E element){
    vector.insertElementAt(element, 0);
}

public E pop(){
    E result = vector.firstElement();
    vector.removeElementAt(0);
    return result;
}
```

### Example (4)

3. Remove the specification of subclass and replace the initialization of the delegate field with a new instance of a superclass object.

```
public class Stack<E> {
    private Vector<E> vector = new Vector<E>();
    ...
}
```

4. Add a method for each method of the superclass used directly by the client application.

```
public int size(){  
    return vector.size();  
}  
  
public boolean isEmpty(){  
    return vector.isEmpty();  
}
```

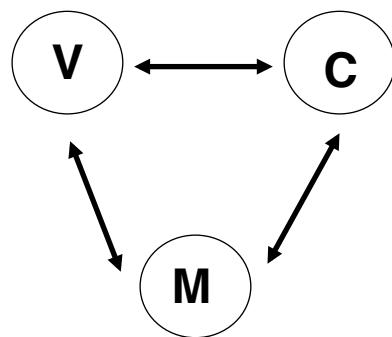
**Title:**

Separate Domain From Presentation

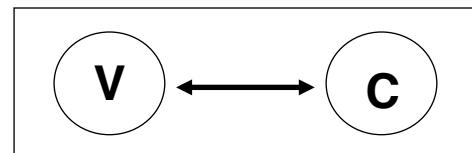
**Target:**

Remove the application logic from a user interface.

## Architecture MVC (1)



## Architecture MVC (2)



Presentation Logic

Domain Logic

## Model extraction

- Application of the refactoring method: “Separate Domain from Presentation”
  - Separation between view and program logic
  - Introduction of the “Observer” pattern

## Observer pattern (1)

### Intent

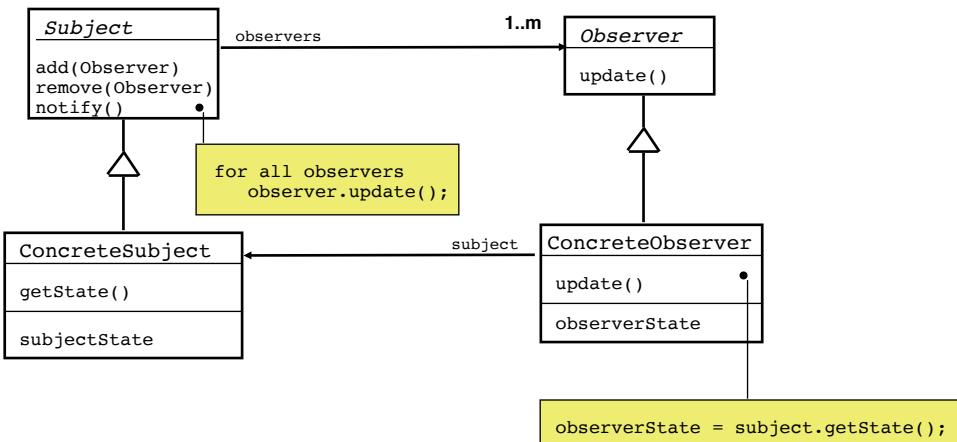
It defines a 1-m dependency among objects so that when one object changes its state, the others are informed.

### Use

- When modifying an object means also modify other ones, but you do not know how many they are
- When this dependency must be dynamic.  
An object must be able to notify its changes, without having to necessarily know the other related objects

## Observer pattern (2)

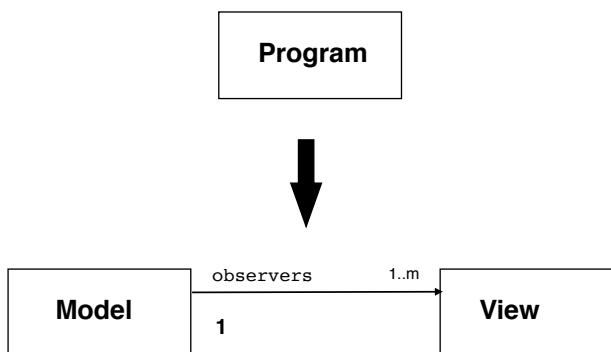
### Structure



## Observer pattern (3)

### Consequences

- Observer and Subject may be changed independently  
-> "favor object composition over class inheritance"
- New observers can be dynamically added, without consequences for subject
- The Observer objects can be part of whatever class hierarchy
- The principle supports the idea of broadcasting

**Scheme****Procedure****Steps**

1. Change the GUI class so that it becomes the observer of the model (separation of observer / observable)

**Compile and test**

2. Isolate the access to the current data model within the GUI using get / set methods

**Compile and test**

3. Create new fields in the model, corresponding to data, also with get / set access methods and move the application's functions into the model

**Compile and test**

**Example (1)**

```

public class Program extends JFrame {
    private JTextField startEntry, endEntry, differenceEntry;

    public Program(String title){
        super(title);
        ...
        endEntry = new JTextField();
        endEntry.addFocusListener(new EndFocus());
        endEntry.addActionListener(new EndAction());
        ...
        ...
    }

    private class EndFocus extends FocusAdapter {
        public void focusLost(FocusEvent event){
            endCheck();
        }
    }

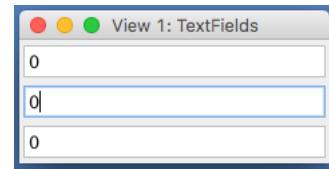
    private class EndAction implements ActionListener {
        public void actionPerformed(ActionEvent event){
            endCheck();
        }
    }
}

```

Sandro Pedrazzini

Refactoring

59

**Example (JavaFX version)**

```

public class Program extends Application {
    private TextField startEntry, endEntry, differenceEntry;

    public void start(Stage primaryStage){
        ...
        endEntry = new TextField();
        endEntry.setOnAction(event -> endCheck());

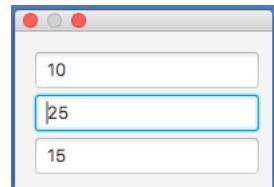
        endEntry.focusedProperty()
            .addListener((observable, oldValue, newValue) -> {
                if (!newValue) {
                    endCheck();
                }
            });
        ...
    }
    ...
}

```

Sandro Pedrazzini

Refactoring

60



**Example (2)**

```

public void startCheck(){
    ...
    computeDifference();
}

public void endCheck(){
    ...
    computeDifference();
}

private void computeDifference(){
    int start = Integer.parseInt(startField.getText());
    int end = Integer.parseInt(endField.getText());
    differenceField.setText(String.valueOf(end - start));
}

private void computeEnd(){
    int start = Integer.parseInt(startField.getText());
    int difference = Integer.parseInt(differenceField.getText());
    endField.setText(String.valueOf(start + difference));
}

```

**First transformation (1)**

Change the GUI class so that it becomes the observer of the model

(You could extend the Java Observable, but in this case we prefer to show a complete realization)

**a) Model creation**

```

public class Model {
    private List<Observer> observers = new ArrayList<Observer>();

    public void addObserver(Observer observer){
        observers.add(observer);
    }

    protected void notifyObservers(){
        for (Observer observer : observers) {
            observer.update(this);
        }
    }
}

```

## First transformation (2)

Each GUI container must be defined as *Observer*, in order to be able to answer to the *update()* message.

```
public interface Observer {  
    void update(...);  
}
```

## First transformation (3)

### b) Reference on the GUI

```
public class Win extends JFrame implements Observer {  
    private JTextField startField, endField, differenceField;  
    private Model model; //managed reference here  
  
    public Win(String title, Model model){  
        super(title);  
        ...  
        this.model = model;  
        initViewFromModel() //initialized through the model  
    }  
  
    private void initViewFromModel(){  
        model.addObserver(this);  
        update();  
    }  
  
    public void update(){  
    }
```

## JDK Observer interface (*not used*)

public interface **Observer**

A class can implement the `observer` interface when it wants to be informed of changes in observable objects.

**Since:**

JDK 1.0

**See Also:**

[Observable](#)

### Method Summary

void	<a href="#"><b>update</b>(<code>Observable o, Object arg</code>)</a>
------	--

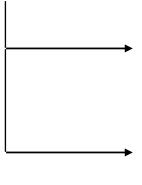
This method is called whenever the observed object is changed.

## Second transformation (1)

Isolate the access to the current data model within the GUI using get / set methods

### a) GUI: Create getters and setters

```
endField.getText()
```



```

public String getEnd(){
    return endField.getText();
}

public void setEnd(String end){
    endField.setText(end);
}

```

## Second transformation (2)

**b) GUI: substitute the direct access**

```

protected void computeDifference(){
    int start = Integer.parseInt(getStart());
    int end = Integer.parseInt(getEnd());
    setDifference(String.valueOf(end - start));
}

protected void computeEnd(){
    int start = Integer.parseInt(getStart());
    int difference = Integer.parseInt(getDifference());
    setEnd(String.valueOf(start + difference));
}

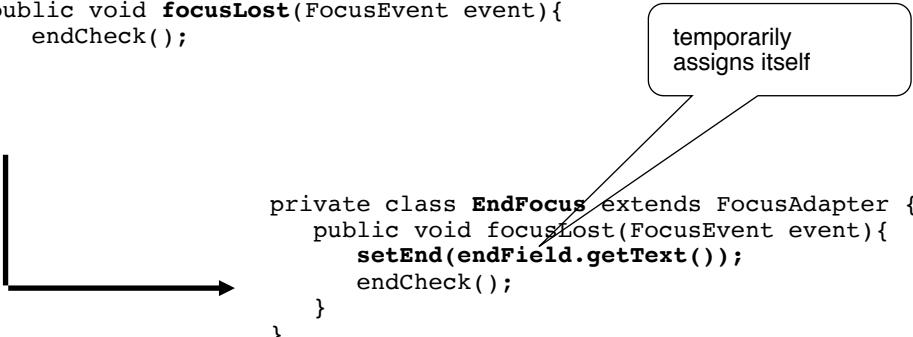
```

## Second transformation (3)

**c) GUI: Update with the user's input**

```

private class EndFocus extends FocusAdapter {
    public void focusLost(FocusEvent event){
        endCheck();
    }
}


private class EndFocus extends FocusAdapter {
    public void focusLost(FocusEvent event){
        setEnd(endField.getText());
        endCheck();
    }
}

```

## Second transformation (4)

### c) GUI: Update with the user's input

```

private class EndAction implements ActionListener {
    public void actionPerformed(ActionEvent event){
        endCheck();
    }
}

private class EndAction implements ActionListener {
    public void actionPerformed(ActionEvent event){
        setEnd(endField.getText());
        endCheck();
    }
}

```

temporarily  
assigns itself

## Third transformation (1)

**Create new fields in the model, corresponding to data, also with get / set access methods and move the application's functions into the model**

### a) Model: new fields

```

public class Model {

    private List<Observer> observers = new ArrayList<Observer>();
    private String start, end, difference;

    public Model(){
        ...
    }

    ...
}

```

## Third transformation (2)

**b) Model: access methods**

```

public String getEnd(){
    return end;
}

public void setEnd(String end){
    this.end = end;
    notifyObservers();
}

```

**c) Move logic from GUI to model**

```

protected void computeDifference(){
    ...
    //no change
}

protected void computeEnd(){
    ...
}

```

## Third transformation (3)

**d) GUI: update access methods**

```

public String getEnd(){
    return endField.getText(); →
}
public String getEnd(){
    return model.getEnd();
}

```

```

public void setEnd(String end){
    endField.setText(end); →
}
public void setEnd(String end){
    model.setEnd(end);
}

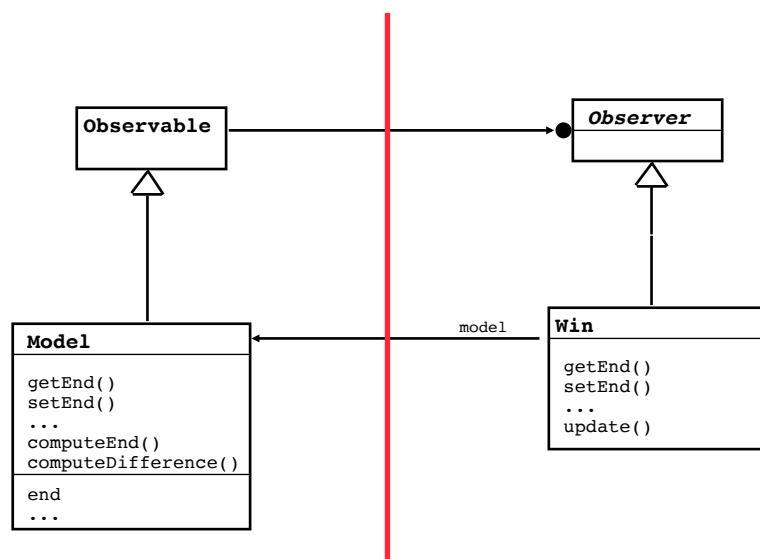
```

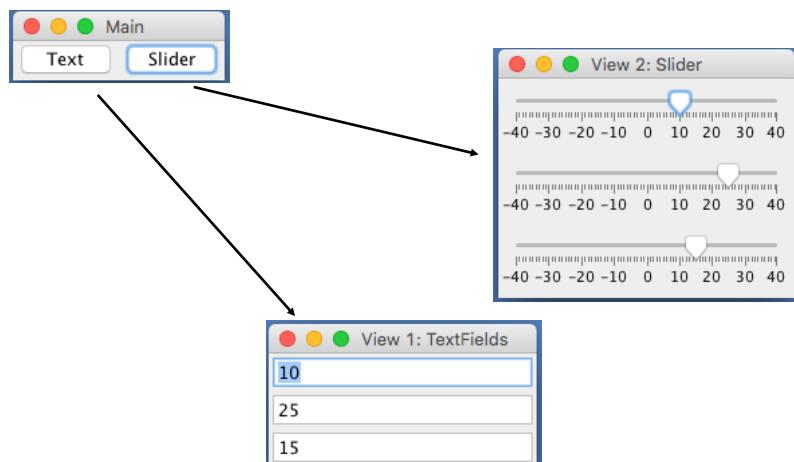
## Third transformation (4)

e) GUI: observer's reaction to the model change's notification

```
public void update(){
    startField.setText(getStart());
    endField.setText(getEnd());
    differenceField.setText(getDifference());
}
```

## Scheme



**Demo**

# Software Engineering, DI

## 1. Dependency Injection

Given following class *Car*:

```
public class Car {
    private IEngine engine;
    private IBody body;
    private Frame frame;
    private Interior interior;
    private Wheel[] wheels;

    public Car() {
        engine = new Engine();
        body = new Body();
        ...
    }
    ...
    public double getWeight() {
    }
}
```

1. Integrate the Guice library into your project (download the jar or specify Maven dependency).

```
<dependency>
    <groupId>com.google.inject</groupId>
    <artifactId>guice</artifactId>
    <version>3.0</version>
</dependency>
```

2. Implement and add further classes, in order to get a more complete description of *Car*.
3. Invert the class dependencies (make use of DI, using constructor or `@Inject` fields, passing the dependencies from outside)
4. Install and add to the project the jar files of the Guice library, and write a Guice configuration module to manage the dependencies of the production code (main application).
5. To create the *Wheel* array you need to specify a `@Provide` method.
6. Consider that we need to write a test for the behavior of the method `getWeight()`, the sum of all components's weight.  
Consider also, however, that the weight of the engine and the weight of the body depend on complex configurations, depending on production's year, security rules, etc. and are only available online.  
Write a unit test for the computation of the total weight using two fake classes for *Engine* and *Body*, only used for tests. You need a Guice test configuration module injecting such classes into the *Car* used for test.

7. Adapt the current version adding a second level of dependencies, adding to the wheels the dependency from the tires (class *Tire*), which can have two versions: Winter and Summer version, having an effect on weight. Manage such choices within the configuration module (you will need a second configuration module, or you will need to call the method “`requestInjection()`”, in order to force the dependency injection into the wheels.

## Deepening

### Dependency Injection

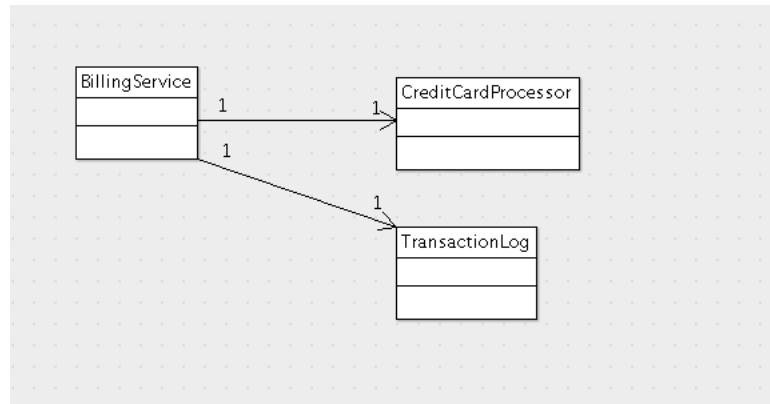
## Dependency Injection (DI)

- Way of configuring an object, where the object dependencies are specified by external entities
- The alternative is that the object defines by itself, internally, the dependencies
- Also called Inversion of Control (IoC), which, however, has a wider meaning (the principle of the framework in general)

## Example

- Class managing the execution of a payment (*BillingService*)
- Both, the class responsible for the processing of credit card (*CreditCardProcessor*), and the class responsible for writing the log information (*TransactionLog*) must be modifiable

## Example (2)



## Example (3)

- **BillingService** charges the order to the credit card. The transaction will be recorded (log) in case of success and in case of failure

```
public interface IBillingService {  
  
    Result chargeOrder(Order order, CreditCard creditCard);  
  
}
```

## Example (4)

- Processing scheme

```
public Result chargeOrder(Order order, CreditCard creditCard) {  
    ...  
    Result result =  
        processor.process(order.getAmount(), creditCard);  
  
    transactionLog.logChargeResult(result);  
  
    ...  
    return result;  
}
```

## Example (5)

```

public class BillingService implements IBillingService {

    public Result chargeOrder(Order order, CreditCard creditCard) {
        ICreditCardProcessor processor = new PaypalCreditCardProcessor();
        ITransactionLog transactionLog = new DatabaseTransactionLog();

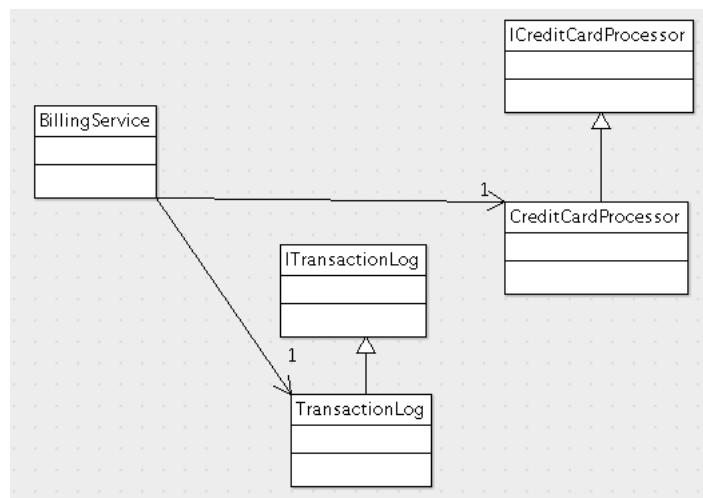
        Result result = processor.process(order.getAmount(), creditCard);
        transactionLog.logChargeResult(result);

        ...
    }
}

```

## Example (6)

### Current dependencies situation

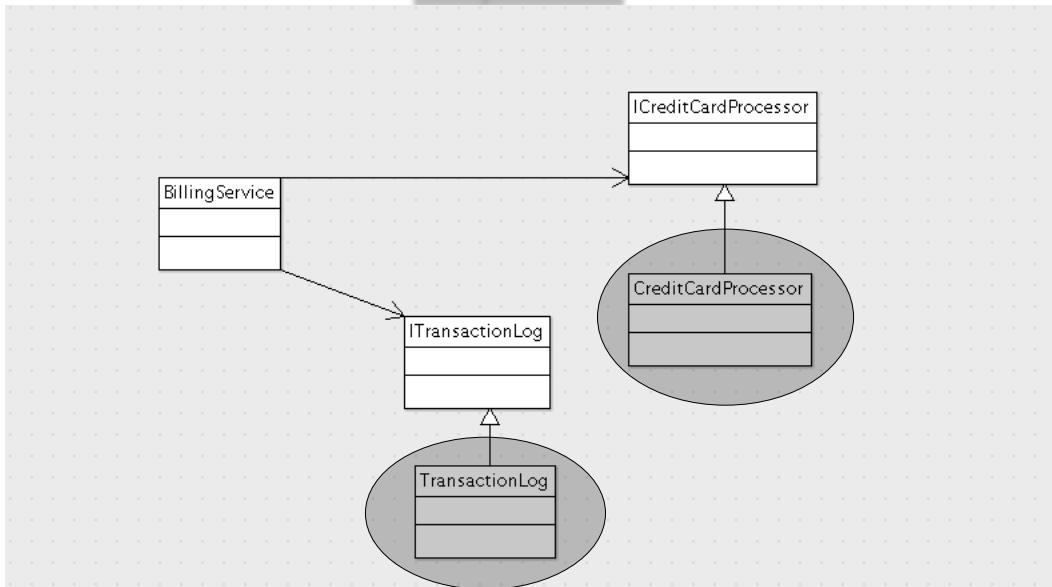


## Comments

- The previous code poses problems of modularity and test
- The direct dependency, at compile time, from the specific credit card processor means that each call to the method, even during the test, will execute a real transaction on your credit card!

```
ICreditCardProcessor processor = new PaypalCreditCardProcessor();
```

## Objective



## Dependency Injection

- The idea is simple: the dependencies are passed from the outside
- With this pattern we make a further step towards the separation between behavior and dependency resolution
- *BillingService* is no longer responsible for the resolution, because the real dependency objects are passed as parameters

## Dependency Injection (2)

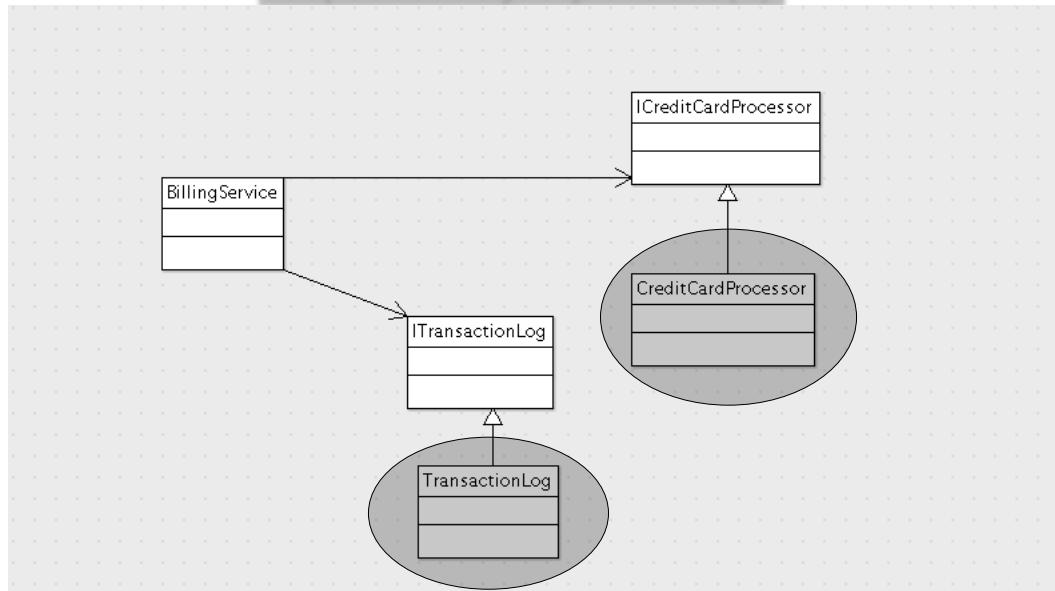
- The dependency objects are provided to *BillingService* from the outside, through the constructor (or through a *set()* method)

```
public BillingService(ICreditCardProcessor creditCardProcessor,  
                     ITransactionLog transactionLog) {  
    this.creditCardProcessor = creditCardProcessor;  
    this.transactionLog = transactionLog;  
}
```

## Dependency Injection (3)

```
public class BillingService implements IBillingService {  
    private ICreditCardProcessor creditCardProcessor;  
    private ITransactionLog transactionLog;  
  
    public BillingService(ICreditCardProcessor creditCardProcessor,  
                         ITransactionLog transactionLog) {  
        this.creditCardProcessor = creditCardProcessor;  
        this.transactionLog = transactionLog;  
    }  
  
    public Result chargeOrder(Order order, CreditCard creditCard) {  
        ...  
        Result result =  
            creditCardProcessor.process(order.getAmount(), creditCard);  
        transactionLog.logChargeResult(result);  
  
        ...  
        return result;  
    }  
}
```

## Dependency Injection (4)



## Unit test

```
public class BillingServiceTest {  
  
    private ICreditCardProcessor processor = new FakeCreditCardProcessor();  
    private ITransactionLog transactionLog = new InMemoryTransactionLog();  
  
    public void testSuccessfulCharge() {  
        Order order = new PizzaOrder(100);  
        CreditCard creditCard = new CreditCard("1234", 11, 2010);  
  
        IBillingService billingService =  
            new BillingService(processor, transactionLog);  
  
        Result result = billingService.chargeOrder(order, creditCard);  
  
        assertTrue(result.hasSuccessfulCharge());  
        assertEquals(100, result.getAmountOfCharge());  
        assertTrue(result.successfullyLogged());  
    }  
}
```

## Comments

- No dependency between BillingService and specific classes
- Easy way to manage dependencies: how does it scale?

## Comments (2)

- By passing all the dependencies to the constructor, for each new dependency you need a new parameter.  
The dependencies are exposed via API
- Also client classes of *IBillingService* must manage the dependencies in the same way

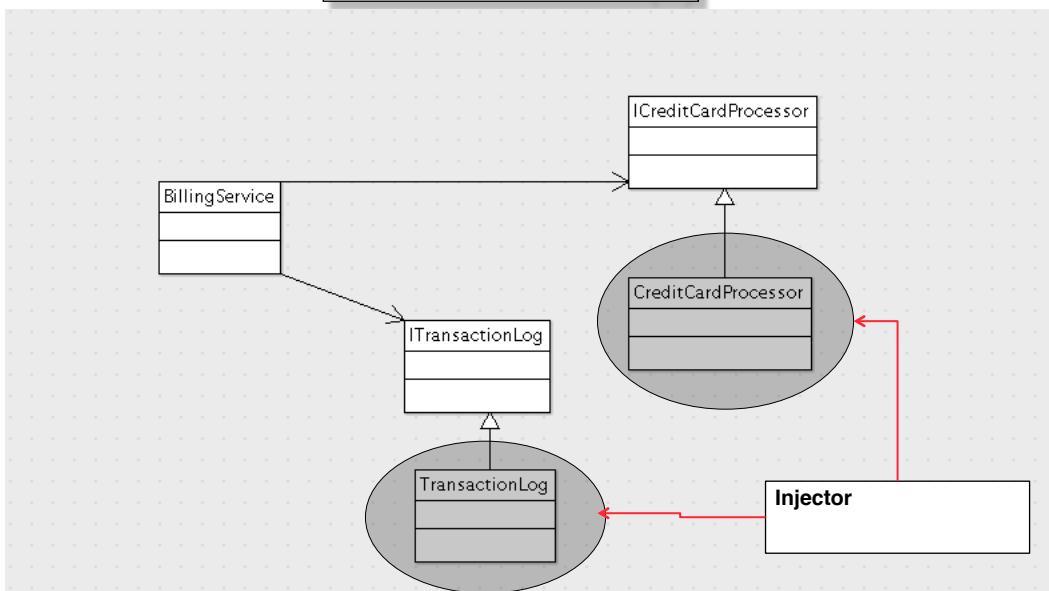
## Generalization

- You can go back in the dependencies chain (classes that depend on an implementation of *IBillingService*)
- These classes will have to provide a parameter for *IBillingService*
- The end point will be the “top-level” classes
- Again: how does it scale? Use a dedicated framework.

## Generalization (2)

- In order to manage the dependencies until the "top-level" classes, it is useful to exploit a framework, which helps to build the dependencies chain
- We have different choices
  - Bean Container di Spring
  - Guice
  - PicoContainer
  - ...

## Generalization (3)



## DI with Guice

- The DI pattern lets you write more modular and more easily testable code
- Frameworks like Guice facilitate the implementation
- They allow you to associate the interfaces with the necessary implementations, dependent on the contexts

## DI with Guice (2)

- The configuration is specified in a Guice module, a class that implements the interface *Module*

```
public class BillingModule extends AbstractModule {  
  
    protected void configure() {  
        bind(ITransactionLog.class).to(DatabaseTransactionLog.class);  
        bind(ICreditCardProcessor.class).to(PaypalCreditCardProcessor.class);  
  
        bind(IBillingService.class).to(BillingService.class);  
    }  
}
```

## DI with Guice (3)

- Inserting @Inject in the constructor of *BillingService*, simply tells Guice to initialize the dependencies as specified in the configuration module.  
**The constructor will never be called explicitly in the code, the object will be constructed by the framework.**
- **Remark:** it is not necessary to specify dependencies as interface/class pairs.  
The relationship class/subclass is also recognized  
If there is only one variant for a class, there is no need to specify the kind of dependency

## DI with Guice (4)

```
public class BillingService implements IBillingService {  
    private ICreditCardProcessor creditCardProcessor;  
    private ITransactionLog transactionLog;  
  
    @Inject  
    public BillingService(ICreditCardProcessor creditCardProcessor,  
                         ITransactionLog transactionLog) {  
        this.creditCardProcessor = creditCardProcessor;  
        this.transactionLog = transactionLog;  
    }  
  
    public Result chargeOrder(Order order, CreditCard creditCard) {  
        ...  
    }  
}
```

## DI with Guice (5)

- **Injectable properties: it is not strictly necessary to provide the constructor for initialization**

```
public class BillingService implements IBillingService {  
    @Inject  
    private ICreditCardProcessor creditCardProcessor;  
  
    @Inject  
    private ITransactionLog transactionLog;  
  
    public Result chargeOrder(Order order, CreditCard creditCard) {  
        ...  
    }  
}
```

## How to use it

```
public static void main(String[] args) {  
    Injector injector = Guice.createInjector(new BillingModule());  
  
    //it receives the top level object  
    IBillingService billingService = injector.getInstance(IBillingService.class);  
  
    Result result = billingService.chargeOrder(  
        new PizzaOrder(100), new CreditCard("1234", 6, 2012));  
  
    if (result.isSuccessfulCharge()) {  
        System.out.println("Receipt value: " + result.getAmountOfCharge());  
    } else {  
        System.out.println("Receipt value: " + result.getErrorMessage());  
    }  
}
```

## Further injection points (2)

- Also a single method can be marked with `@Inject`: Guice will call this method with the associated parameter(s) immediately after calling the constructor.
- It can have more parameters and does not need to be public

```
@Inject  
public void setTransactionLog(ITransactionLog transactionLog){  
    this.transactionLog = transactionLog;  
}
```

## Use of Guice for tests (1)

```
public class TestModule extends AbstractModule {  
  
    protected void configure() {  
        bind(ITransactionLog.class).to(InMemoryTransactionLog.class);  
        bind(ICreditCardProcessor.class).to(FakeCreditCardProcessor.class);  
  
        bind(IBillingService.class).to(BillingService.class);  
    }  
}
```

## Use of Guice for tests (2)

```
public class BillingServiceWithGuiceTest {  
  
    private Injector injector = Guice.createInjector(new TestModule());  
  
    @Test  
    public void testSuccessfulCharge() {  
        Order order = new PizzaOrder(100);  
        CreditCard creditCard = new CreditCard("1234", 11, 2010);  
  
        IBillingService billingService =  
            injector.getInstance(IBillingService.class);  
  
        Result result = billingService.chargeOrder(order, creditCard);  
  
        assertTrue(result.hasSuccessfulCharge());  
        assertEquals(100, result.getAmountOfCharge());  
        assertTrue(result.successfullyLogged());  
    }  
}
```

## Provider

- It happens that for the creation of an object you need more code than the simple call to the constructor
- Usually, in such cases, we would write a factory method that contains a call to the constructor, plus needed code
- In Guice the method is specified within the configuration module and it annotated with `@Provides`
- The corresponding binding in `configure()` is not needed anymore

## Method @Provides

```
public class BillingModule extends AbstractModule {  
  
    protected void configure() {  
        // bind(ITransactionLog.class).to(DatabaseTransactionLog.class);  
        ...  
    }  
  
    @Provides  
    ITransactionLog provideTransactionLog() {  
        DatabaseTransactionLog transactionLog = new DatabaseTransactionLog();  
        transactionLog.setJdbcUrl("jdbc:mysql://localhost/trans");  
        transactionLog.setThreadPoolSize(30);  
  
        return transactionLog;  
    }  
}
```

## Provider class

- As soon as a `@Provides` method is too complex, you can evaluate the option of using a dedicated provider class
- Guice provides for this purpose a *Provider* interface

```
public interface Provider<T> {  
    T get();  
}
```

## Provider class (2)

Class responsible for the creation of the *ITransactionLog* object

```
public class DBTransactionProvider
    implements Provider<ITransactionLog> {

    public ITransactionLog get() {
        DatabaseTransactionLog transactionLog =
            new DatabaseTransactionLog();
        transactionLog.setJdbcUrl("jdbc:mysql://localhost/trans");
        transactionLog.setThreadPoolSize(30);

        return transactionLog;
    }
}
```

## Provider class (3)

- Binding to specify in the configuration module

```
public class BillingModule extends AbstractModule {

    protected void configure() {
        bind(ITransactionLog.class).
            toProvider(DBTransactionProvider.class);
        ...
    }

    ...
}
```

## Provider class (4)

- Combined with anonymous class

```
public class BillingModule extends AbstractModule {

    protected void configure() {
        bind(ITransactionLog.class).
            toProvider(new Provider<ITransactionLog>() {
                public ITransactionLog get() {
                    DatabaseTransactionLog tl =
                        new DatabaseTransactionLog();
                    tl.setJdbcUrl("jdbc:mysql://localhost/trans");
                    tl.setThreadPoolSize(30);

                    return tl;
                }
            });
        ...
    }
    ...
}
```

## toInstance()

- Simply creating the instance within `configure()`

```
public class BillingModule extends AbstractModule {

    protected void configure() {
        DatabaseTransactionLog transactionLog =
            new DatabaseTransactionLog();
        transactionLog.setJdbcUrl("jdbc:mysql://localhost/trans");
        transactionLog.setThreadPoolSize(30);

        bind(ITransactionLog.class).toInstance(transactionLog);
        ...
    }
    ...
}
```

## How to pass primitives

- In configuration()

```
bindConstant(Names.named("pool size")).to(30);
```

- In class constructor

```
@Inject
public DatabaseTransactionLog(@Named("pool size") int size) {
    ...
}
```

## Request Injector

- You need to force the dependencies initialization if you need to create the object yourself, explicitly calling the constructor

```
public class BillingModule extends AbstractModule {
    protected void configure() {
        bind(ITransactionLog.class).to(DatabaseTransactionLog.class);
        bind(ICreditCardProcessor.class)
            .to(PaypalCreditCardProcessor.class);

        IBillingService billingService = new BillingService();
        //necessary to force the injection billingService
        requestInjection(billingService);
        ...
    }
}
```

## Use of more modules

- If you need to use objects of the same class, but initialized differently, you can use several modules simultaneously
- The use of more than one module can also be useful in cascade: if an object of a module must be created manually within a provider, the dependencies may be specified in a second module.

## Use of more modules (2)

```
public class DBTransactionProvider implements Provider<ITransactionLog> {  
    public ITransactionLog get() {  
        Injector injector = Guice.createInjector(new DBProductionModule());  
        ITransactionLog transactionLog =  
            injector.getInstance(ITransactionLog.class);  
        ...  
        return transactionLog;  
    }  
}
```

## Scope

- By default, Guice returns a new instance each time an object is required

- Example

- Every time you request an object of type *ITransactionLog* to this module, a new instance of *InMemoryTransactionLog* is created

```
bind(ITransactionLog.class).  
    to(InMemoryTransactionLog.class);
```

## Scope (2)

- There are other scope you can select:
  - For the entire application life cycle (@Singleton)
  - For a session duration (@SessionScoped)
  - For a single request (@RequestScoped)

- Example

- Scope configurated during the binding:

```
bind(ITransactionLog.class).  
    to(InMemoryTransactionLog.class).  
    in(Singleton.class);
```

## Scope (3)

- **Eager singleton**

- **Guice provides the way of specifying singleton objects to be built in advance (eagerly)**

```
bind(ITransactionLog.class).  
    to(InMemoryTransactionLog.class).  
    asEagerSingleton();
```

# Software Engineering: AOP

## 1. Method Interceptor with Guice: tracer

Given the program developed in a previous lab, using *Composite* and *Visitor* to realize the different visualization functions of an organizational unit, realize, with Guice “method interceptor”, an aspect allowing to trace on standard error, the time (in milliseconds) used by every call to the *Visitor* methods.

Here is a possible version of *Visitor*:

```
public interface Visitor {  
    void visitOffice(Composite composite);  
    void visitCompany(Composite composite);  
    void visitPerson(Person person);  
}
```

The aspect should be inserted into a class named *TracingInterceptor*, implementing *MethodInterceptor*, which must be activated through a *bindInterceptor()* from a Guice configuration module, which should bind the *TracingInterceptor* to any method of any *Visitor* subclass.

## Deepening

### **Aspect Oriented Programming (AOP)**

### **Aspect Oriented Programming**

- Programming paradigm, which involves the insertion of common functionality (cross-cutting concerns) to existing functionality
- The aim is to separate the common aspects (logging, security, exception handling, etc..) from real business logic

## Aspect Oriented Programming (AOP)

- All paradigms, especially the OO paradigm, provide means for the encapsulation of functionalities into separate entities, creating abstractions (methods, classes).
- There are, however, features that may not fit easily into such an organization, because they are more transversal (cross-cutting)
- **Example:** logging

## Typical cross-cutting concerns

- **Synchronization**
- **Memory management**
- **Persistency**
- **Security**
- **Caching**
- **Logging**
- **Monitoring**
- **Business rules**
- ...

## Implementations

- The most known Java implementations are AspectJ e Hyper/J
- AOP is integrated in various frameworks, especially through the implementation of AspectJ
  - Spring
  - JBoss AOP
  - Guice

## Basis terminology

- **Cross-cutting concern**

Common functionality (transversal) to be shared among multiple functions.

Example:

logging functionality, identical, within different methods or classes, in which, for example, a log must be executed entering and exiting the method call.

## Basis terminology (2)

- **Advice**

Additional code to be applied (added) to existing business model.

Example:

Unique logging code, to be called separately, when needed.

## Basis terminology (3)

- **Pointcut**

Execution point within the application. The point at which a cross-cutting concern should be applied.

Example:

Point reached when a certain method begins (entry logging) or exits (exit logging)

## Basis terminology (4)

- **Aspect**

Module that combines the pointcut (join point) and the advice body descriptions.

Example:

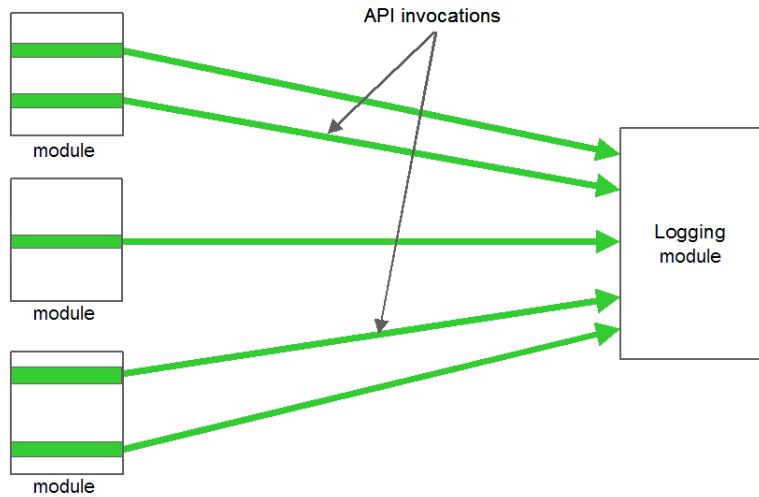
We add an aspect to the application, specifying the logging mode and when this should be executed

## Logging

- An aspect functionality like logging can be found in many parts of the code, making it difficult code understanding and code maintenance
- A change to logging code, can have an impact in different modules, classes and methods

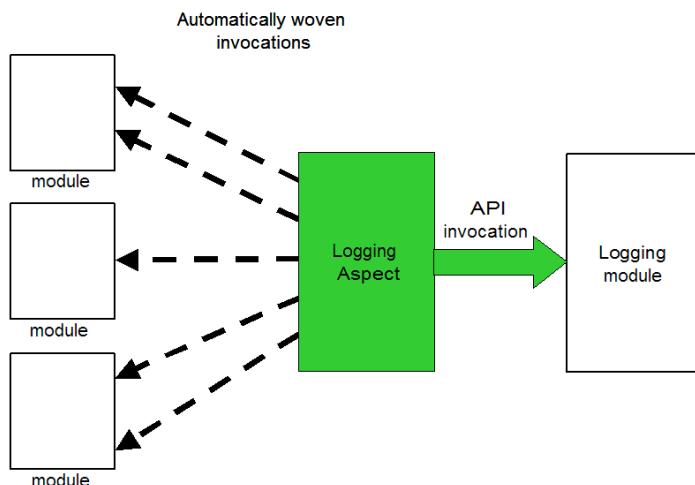
## Logging (2)

- Code in every module



## Logging (3)

- Centralization within a unique aspect



## Example

- Payment with credit card

```
public class BillingService implements IBillingService {  
    ...  
  
    public Result chargeOrder(Order order, CreditCard card) ...{  
        Result result =  
            creditCardProcessor.process(order.getAmount(), card);  
  
        ...  
        return result;  
    }  
}
```

## Example (2)

- We also need security control and logging

```
public Result chargeOrder(User user, Order order, CreditCard card)  
    throws Exception {  
    logger.debug("Start method " +  
        new Object(){}.getClass().getEnclosingMethod().getName());  
  
    if (!checkUserPermission(user)) {  
        logger.info("User not authorized...");  
        throw new UnauthorizedException();  
    }  
  
    Result result =  
        fCreditCardProcessor.process(order.getAmount(), card);  
  
    logger.debug("End method " + ...);  
  
    return result;  
}
```

## Example (3)

- In the previous code, logging and security elements must be considered cross-cutting concerns
- What happens if we have to modify security elements in the application?  
Since these elements are scattered throughout the application, the changes will be several.
- It would be useful to manage them centrally

## Aspect

- AOP encourages you to resolve the problem of the presence of individual elements in most parts of the code allowing you to express these cross-cutting concerns through aspects
- An aspect contains an advice (code to execute) and a pointcut (statement of when the advice should be executed)
- **Example:** an aspect could contain
  - Security code
  - Command that specifies that the check is performed every time before the call to process()

## Aspect (2)

- Possible example using AspectJ “pseudocode”

```
public aspect SecurityCheck {  
  
    before() : within(Receipt IBillingService.chargeOrder(  
                           User user, Order order, CreditCard card)) &&  
               call(ChargeResult ICreditCardProcessor.process(  
                           long, CreditCard)) {  
  
        if (!checkUserPermission(user) {  
            logger.info("User not authorized...");  
            throw new UnauthorizedException();  
        }  
    }  
}
```

## Aspect (3)

- *AspectJ* provides a series of declarative commands to specify the *join points*
- The *join points* are the most critical element, because through their representation, also expressed through regular expressions, you specify the match
- Complex expressions make the code less "predictable" and therefore not maintainable

## Pointcut examples

- **execution(\* set\*(\*))**

Match with the execution of all methods whose name starts with "set" and have a single parameter of any type

- **within(ch.supsi.\*)**

It restricts the scope of the pointcut to anything (class, method) in the package "ch.supsi"

- **this(CreditCard)**

This pointcut resolves when the current running object (this) is an instance of the CreditCard class

- **execution(\* set\*(\*)) && within(ch.supsi.\*)) && this(CreditCard)**

Combination of the three previous criteria

## Implementations

- **There are basically two ways to implement AOP**

- **Class-waving**

It injects the implementations of the single aspects directly into the classes where they must be executed. The weaving can be applied at compile-time, as well as load or run-time.

- **Proxy**

Through the use of Proxy, the method call to an object is intercepted at runtime, executing the needed AOP functionality.

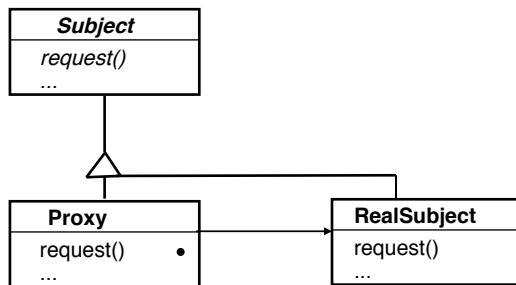
Usually Java dynamic proxy or CGLIB (code generation lib) are used to implement this solution.

The framework using the Proxy are generally simpler and their implementations are often based on the method interceptor mechanism.

## Implementation: proxy (1)

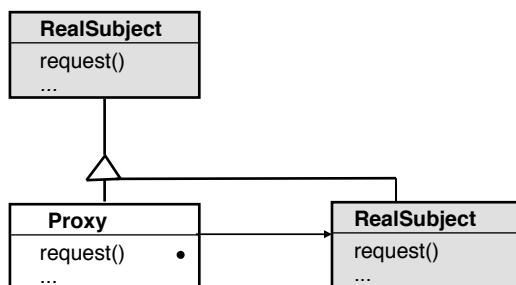
- A new subclass is generated, extending the base class.

The generation of the subclass is performed dynamically, at runtime (Java dynamic proxy), or modifying the bytecode during its generation (CGLIB).



## Implementation: proxy (2)

- The Java dynamic proxy can be applied only if there is a specific interface in the class to be extended.
- With CGLIB, however, the proxy inherits from the same class of which it is also a proxy.



## AOP in frameworks

- **Frameworks like Spring, Guice or others, integrate *AspectJ* functionalities, creating, if possible, a further abstraction level.**
  - **Spring:** it simplifies its use, maintaining, however, the possibility to directly access advanced and specific *AspectJ* features.
  - **Guice:** it simplifies and limits its use, with the aim of promoting its use where this helps improving the code, but avoiding the use of complex features that would have bad consequences on the maintenance.

## AOP with Guice

- Guice provides AOP through the “method interception” mechanism
- In this way you can execute an *advice* each time a specific method is invoked.
- **AOP terminology in Guice:**
  - **Matcher:** element executing the match => pointcut
  - **Method Interceptor:** part of code to be executed => advice

## AOP with Guice (2)

- **Matcher**

Declaration allowing to accept or refuse a value.

In **Guice** you need two matchers: one to specify the class and one to specify the method.

- **Method Interceptor**

Code executed when a method resolving a match is invoked. It receives information and references on the intercepted method: the method itself, its arguments, and the invoking object.

## Example

- We reuse the example of the credit card payment already analyzed with DI.

- **Request:**

Implement a check using AOP, called during the payment execution, to verify if the payment can be really executed, considering the days of the week as constraints (no payments on Sunday)

There must be one or more days in the week in which the payment service cannot be used.

## Example (2)

- Current code

```
public class BillingService implements IBillingService {  
    ...  
  
    public Result chargeOrder(Order order, CreditCard card) {  
        Result result =  
            processor.process(order.getAmount(), card);  
  
        ...  
        return result;  
    }  
}
```

## Matcher

- We define a new annotation to specify the Matcher.  
It will be used to signal the interested method.

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface WeekDayCheck {  
}
```

## Matcher (2)

- Use of the annotation

```
public class BillingService implements IBillingService {
    ...
    @WeekDayCheck
    public Result chargeOrder(Order order, CreditCard card) {
        Result result =
            processor.process(order.getAmount(), card);
        ...
        return result;
    }
}
```

## Interceptor

**This code is executed each time a match is found.  
The matched method is called within this interceptor  
(*invocation.proceed()*)**

```
public class SundayBlocker implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable {
        Calendar today = new GregorianCalendar();
        String todayDisplayName = today.getDisplayName(Calendar.DAY_OF_WEEK,
                                                       Calendar.LONG,
                                                       Locale.ENGLISH);
        Order order = (Order) invocation.getArguments()[0];
        if (todayDisplayName.equals("Sunday")) {
            throw new IllegalStateException(
                invocation.getMethod().getName() + " for " + order.getAmount()
                + " not allowed on " + todayDisplayName);
        }
        return invocation.proceed();
    }
}
```

## Interceptor (2)

- **Real intercepted method call**

```
invocation.proceed();
```

- **Access to the parameters values**

```
invocation.getArguments()
```

- **Access to the intercepted method information**

```
invocation.getMethod()
```

## Binding module

- **Guice uses a module to specify the bindings, and we use it to bind the interceptor to the matcher.**

```
public class NotOnSundayModule extends AbstractModule {  
    protected void configure() {  
        bindInterceptor(Matchers.any(),  
                        Matchers.annotatedWith(WeekDayCheck.class),  
                        new SundayBlocker());  
  
        bind(ITransactionLog.class).  
            toProvider(DatabaseTransactionProvider.class);  
        bind(ICreditCardProcessor.class).  
            to(PaypalCreditCardProcessor.class);  
  
        bind(IBillingService.class).to(BillingService.class);  
    }  
}
```

## Binding module (2)

```
bindInterceptor(
    Matchers.any(),
    Matchers.annotatedWith(WeekDayCheck.class),
    new SundayBlocker());
```

Match with any class

Only with methods with  
this annotation

Object containing the  
interceptor method

## Main

```
public class Main{
    public static void main(String[] args) {
        Injector injector =
            Guice.createInjector(new NotOnSundayModule ());
        ...
        IBillingService billingService =
            injector.getInstance(IBillingService.class);
        ...
    }
}
```

## Some matchers Methods

```
static Matcher<AnnotatedElement> annotatedWith(Annotation annotation)
    Returns a matcher which matches elements (methods, classes, etc.) with a given annotation.

static Matcher<Object> any()
    Returns a matcher which matches any input.

static Matcher<Object> identicalTo(Object value)
    Returns a matcher which matches only the given object.

static Matcher<Class> inPackage(Package targetPackage)
    Returns a matcher which matches classes in the given package.

static Matcher<Class> inSubpackage(String targetPackageName)
    Returns a matcher which matches classes in the given package and its subpackages.

static Matcher<Object> only(Object value)
    Returns a matcher which matches objects equal to the given object.

static Matcher<Class> subclassesOf(Class<?> superclass)
    Returns a matcher which matches subclasses of the given type (as well as the given type).
```

## Another example of binding module(2)

### No need of annotations

```
bindInterceptor(
    Matchers.subclassesof(BillingService.class),
    Matchers.any(),
    new SundayBlocker());
```

Match this class and its subclasses

All methods

Object containing the interceptor method

## Example 2: security

- **Request**

Check if an user is trying to execute a certain operation and if he has the right to do it.

- **Approach**

The actions a user is allowed to execute are specified following a “role” based approach.

Each user can have different roles, each of which allows a certain number of actions.

## Example 2: security (2)

- We specify, using annotations, to which roles we assign the right to execute a certain action.
- An **interceptor** is created to verify if the invoking user has the correct role to execute the method  
(in a real case, role and action would be bound to permissions, so we would also need to verify them)

## Roles

- To specify the roles in a simple and, however, controlled way, we use enumerations:

```
public enum Role {  
    CUSTOMER,  
    EMPLOYEE  
}
```

## User Manager

- The User Manager object is used to keep information on the registered users:

```
public interface IUserManager {  
    void setCurrentUser(User user);  
    User getCurrentUser();  
}
```

## User Manager (2)

- Within a multiuser application the User Manager would use the Session object to manage the registered users.
- Within a single user desktop application, however, you can use a singleton
- We can represent the user as a simple single class that manages user name (minimal information) and a set of roles, bound to that user.

## User

```
public class User {  
    private String name;  
    private Set<Role> roles;  
  
    public User(String name, Set<Role> roles) {  
        this.name = name;  
        this.roles = roles;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public Set<Role> getRoles() {  
        return roles;  
    }  
  
    public boolean hasRole(Role role) {  
        return roles.contains(role);  
    }  
    ...  
}
```

## Actions' annotations

- The annotations allow us to tag which roles are required to execute a certain operation
- Used to tag a method. The information is available at runtime
- The `value()` is used to specify the necessary role for that action

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface RequiresRole {

    Role value();
}
```

## Actions

- Imagine now to have a class with some actions, the annotations allow us to specify with which role these actions can be executed

```
public class VideoRental {

    ...

    @RequiresRole(Role.CUSTOMER)
    ...

    @RequiresRole(Role.EMPLOYEE)
    ...
}
```

## Actions (2)

```
public class VideoRental {

    @Inject
    IUserManager userManager;

    @RequiresRole(Role.CUSTOMER)
    public void rentMovie(long movieId) {
        System.out.println(String.format(
            "Movie %d rented by user %s.",
            movieId, userManager.getCurrentUser()));
    }

    @RequiresRole(Role.EMPLOYEE)
    public void registerNewMovie(String name) {
        System.out.println(String.format(
            "New movie \"%s\" registered by user %s.",
            name, userManager.getCurrentUser()));
    }
}
```

## Interceptor

- Now we need an interceptor that uses information specified through the annotation to verify the correct roles

```
public class RoleValidationInterceptor implements MethodInterceptor {

    @Inject
    private IUserManager userMgr;

    public Object invoke(MethodInvocation invocation) throws Throwable {
        Role requiredRole =
            invocation.getMethod().getAnnotation(RequiresRole.class).value();

        if (!userMgr.getCurrentUser().hasRole(requiredRole)){
            throw new IllegalStateException("...");
        }

        return invocation.proceed();
    }
}
```

## Module

- Moreover, we must specify when the interceptor must be used (bind interceptor and matching)

```
public class ExampleModule extends AbstractModule {
    public void configure() {
        bind(IUserManager.class).to(UserManager.class).in(Scopes.SINGLETON);
        RoleValidationInterceptor roleValidationInterceptor =
            new RoleValidationInterceptor();
        bindInterceptor(any(),
            annotatedWith(RequiresRole.class),
            roleValidationInterceptor);

        //necessary, to resolve @Inject within roleValidationInterceptor
        requestInjection(roleValidationInterceptor);
    }
}
```

## Module (2)

- We first associate a `IUserManager` implementation to the interface. Defining it as singleton, we specify that the same object injected into `VideoRental` is also used and injected into `RoleValidationInterceptor`
- Then, we associate the interceptor to all methods annotated with `RequiresRole`
- The last step (`requestInjection()`) is necessary in order to inject into the interceptor (which also uses DI) its dependencies (in our case the `UserManager` object)

## Qualità nei processi di sviluppo

### Integrazione continua

### Integrazione continua

- Pratica dello sviluppo software
- Ogni membro del gruppo di sviluppo integra il proprio lavoro il più frequentemente possibile, solitamente almeno una volta al giorno
- Ogni integrazione viene verificata da un *build* automatico, che lancia i test e verifica eventuali problemi di integrazione

## Integrazione continua (2)

- Molti team di sviluppo trovano che una pratica del genere serve a ridurre al minimo i problemi di integrazione
- Permette al team di sviluppare software coeso in modo molto più rapido
- Aiuta a ridurre i rischi legati allo sviluppo e all'integrazione

## Motivazione

- In molti processi software l'integrazione richiede giorni, settimane o mesi
- Questo perché l'integrazione viene eseguita come ultimo passaggio prima di andare in produzione
- Per poter andare in produzione più frequentemente bisogna trasformare l'integrazione in qualcosa di automatico (integrazione vista come “non-event”)

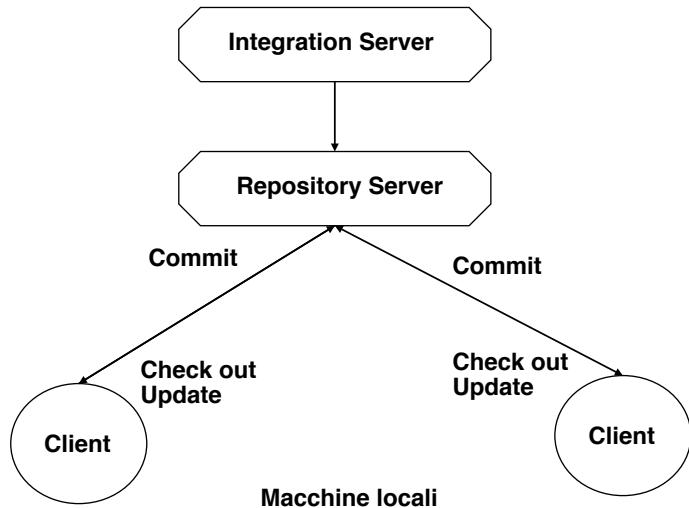
## Motivazione (2)

- Integrando regolarmente significa che ogni sviluppatore ad ogni iterazione aggiunge solo alcune ore di lavoro al progetto integrato
- Anche in questo caso (come nelle altre pratiche di XP) si tratta semplicemente di applicare in modo coerente alcune “buone abitudini”.
- Più frequentemente si integra e più l’integrazione diventa un “non-event”

## Tool

- Integrazione continua è prima di tutto una buona abitudine, perciò di base non necessitano tool particolari
- Ne esistono però alcuni (esempi: **CruiseControl**, **TeamCity**, **Jenkins**) che combinati a un sistema di gestione concorrente di sorgente (**CVS**, **Subversion**, **GIT**, ecc.) e a un sistema di build indipendente da ogni IDE (esempio: Ant, Maven, Gradle) permettono di attivare le verifiche di integrazione in modo automatico.

## Tool (2)

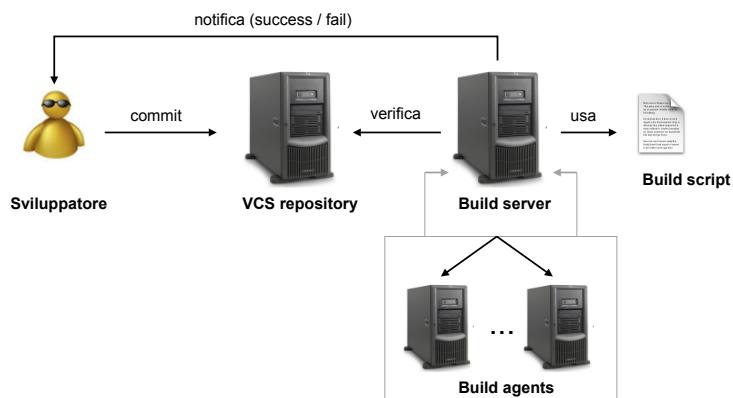


Sandro Pedrazzini

Integrazione continua

7

## Processo



Sandro Pedrazzini

Integrazione continua

8

## Tool (3)

- Esempio di build con Ant

➤ ant integrate

```
Buildfile: build.xml
clean:
all:
compile-src:
compile-tests:
integrate-db:
run-tests:
run-inspections:
delivery:
deploy:
BUILD SUCCESSFUL
Total time: 6 minutes 15 seconds
```

## Tool (4)

- Esempio di commit con Subversion (via linea di comando)

➤ svn commit -m "Aggiunta verifica della connessione"

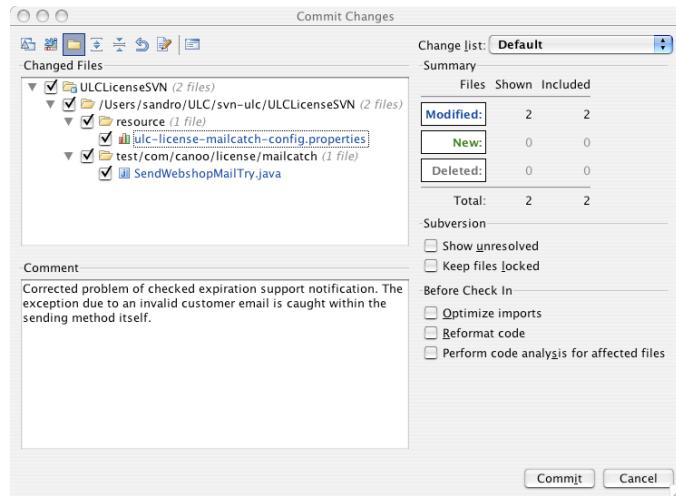
```
Sending src/com/canoo/db/dao/DBServicesImpl.java
Transmitting file data .


```

```
Committed revision 125
```

## Tool (5)

- Esempio di commit integrato in IDE



Sandro Pedrazzini

Integrazione continua

11

## Tool (6)

- Confronto fra versioni

```

10351     mailContent.append(" no new expired support license ");
    } else {
        content.append("Expired support licenses: " + checkedSupportLicenses.size() + "\n");
        fillMailContent(mailContent, bucketsOfExpiredLicenses);
        logger.info("Send summary mail to " + baseConfig.EMAIL_ADMIN);
        MailHandler.sendEmail(baseConfig.EMAIL_ADMIN, null, null, baseConfig.EMAIL_EXP_SUPPORT);
        if (baseConfig.MAIL_BE_SENT_TO_CUSTOMER) {
            sendNotificationToSingleClients(bucketsOfExpiredLicenses);
        }
        return (SupportLicensee[]) checkedSupportLicenses.toArray(new SupportLicense[checkedSupportLicenses.size()]);
    }
}

private static void sendNotificationToSingleClients(Map expirationBuckets) throws Exception {
    if (expirationBuckets.size() == 0) {
        logger.warn("No new support license for each single client");
        Set sortedKeys = expirationBuckets.keySet();
        Iterator iter = sortedKeys.iterator();
        while (iter.hasNext()) {
            List contentList = (List) expirationBuckets.get(iter.next());
            ProductAndSupportLicensePair licensePair = (ProductAndSupportLicensePair) contentList.get(0);
            String mailCustomerText = prepareCustomerText(licensePair, contentList);
            MailHandler.sendSimpleMessage(licensePair.getClientEmail(), prepareSellerAddress(licensePair));
        }
    }
}

private static String prepareSellerAddress(String secondContactEmail) {
    if (baseConfig.RESELLER_AB_CO) {
        if (secondContactEmail != null && secondContactEmail.equals("")) {
            return secondContactEmail;
        }
    }
    return null;
}

private static String prepareCustomerText(ProductAndSupportLicensePair licensePair, List contentList) {
    StringBuffer text = new StringBuffer();
    text.append("Dear ").append(licensePair.getFirstName()).append(" ").append(licensePair.getLastName());
    contentList.iterator();
    Iterator iter = contentList.iterator();
    while (iter.hasNext()) {
        ProductAndSupportLicensePair pair = (ProductAndSupportLicensePair) iter.next();
        text.append(pair.getCompanyName());
        text.append(" (").append(pair.getLicensee().getExtendedNumber()).append(")\n");
    }
    text.append(baseConfig.CUSTOMER_MAIL_TEXT);
    return text.toString();
}

private void fillBucketOfRelatedExpirationLicenses(Map expirationBucket, SupportLicensee supp
ProductLicensee) {
    RelatedProductLicenses relatedProductLicenses = dataservice.loadLicenses(supportLicense);
    for (int i = 0; i < relatedProductLicenses.size(); i++) {
        Client client = relatedProductLicenses[i].getClient();
        String key = client.getEmail() + client.getCompany() + supportLicense.getExpirationDate();
        if (client2 != null) {
            key = key + client2.getEmail() + client2.getCompany(); // singleLicense.getsize();
        }
        List contentList = (List) expirationBucket.get(key);
        if (contentList == null) {
    
```

6 differences Deleted Changed Inserted

Sandro Pedrazzini

Integrazione continua

12

## Tool (8)

- Mail ricevuta da Cruise Control dopo il build

### BUILD COMPLETE - build.180

Date of build: 01/07/2008 16:21:50  
Time to build: 10 seconds  
Last changed: 01/07/2008 16:16:51  
Last log entry: Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Modifications since last successful build: (6)	
modified	sandro /ULCLicense/trunk/src/com/canoo/license/base/data/transaction/TransactionContext.java
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
added	sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification-3.xml
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified	sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification.xml
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified	sandro /ULCLicense/trunk/src/com/canoo/license/base/support/CheckExpiringSupport.java
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified	sandro /ULCLicense/trunk/build.xml
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified	sandro /ULCLicenseTemplates/trunk/ULCBase/ulc-support-additions.sql
	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Sandro Pedrazzini

Integrazione continua

13

## Tool (9)

### Progetti in TeamCity

Welcome, sandro Logout

Projects My Changes Agents (1) Build Queue (0) Administration My Settings & Tools Configure Visible Projects

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- **CobraWunelli Maintenance Build Configuration** Idle Run ↗
  - #build.563-2 Tests passed: 159
  - No artifacts Changes (1) 10 Feb 17:56 (7m:37s)
- **CobraWunelli Trunk Build Configuration** Idle Pending (1) Run ↗
  - #build.680 Tests passed: 191
  - No artifacts Changes (1) 25 Feb 15:33 (12m:09s)

Sandro Pedrazzini

Integrazione continua

14

## Tool (10)

- History di TeamCity

The screenshot shows the TeamCity web interface for the project 'CobraWunelli'. The top navigation bar includes 'Projects', 'My Changes', 'Agents (1)', 'Build Queue (0)', 'Administration', 'My Settings & Tools', and a search bar. The main content area displays the 'History' tab of the build configuration 'CobraWunelli Trunk Build Configuration'. It shows a table of recent builds with columns for '#', 'Results', 'Artifacts', 'Changes', 'Started', 'Duration', 'Agent', 'Tags', and a 'Pin' icon. The table lists builds from #build.680 to #build.666. Build #680 is green (Tests passed: 191). Builds #678, #677, #676, #672, #671, #670, and #669 are green (Tests passed: 185, 172, 172, 175, 175, 175, 175). Build #668 is red (Tests failed: 1 (1 new), passed: 171). Build #666 is red ([Execution timeout] Tests passed: 171). The status bar at the bottom indicates 'Integrazione continua' and the page number '15'.

## Esempio (1)

- Supponiamo di voler aggiungere una nuova funzionalità ad un programma esistente

- Come primo passaggio devo scaricare la versione più aggiornata sulla mia macchina locale (check-out o update, nel caso avessi già una versione locale non aggiornata)
- Quando ho la versione sulla mia macchina, posso iniziare a fare le aggiunte desiderate
- Questo significa non solo aggiungere funzionalità, ma anche adattare vecchio codice, aggiungere e adattare test

## Esempio (2)

- Appena terminato con la nuova funzionalità e i vari test, posso creare il build sulla mia macchina locale
- Questo significa ricompilare tutto, creare e includere le varie librerie, eseguire i test
- Posso considerare di aver terminato il lavoro sulla macchina locale solo quando tutto funziona senza errori
- Ora che il build locale funziona, posso pensare di eseguire un “commit” sul repository comune

## Esempio (3)

- Non ho ancora finito: a questo punto si tratta di eseguire il build sulla macchina di integrazione
- Solo se anche questo build ha successo, si può affermare che i cambiamenti eseguiti fanno parte dell’applicazione
- Il build di integrazione può essere eseguito manualmente oppure automaticamente con tool come TeamCity, che, eseguendo un polling continuo sul repository per determinare se ci sono stati aggiornamenti, fa partire il build quando necessario

## Conflitti (1)

- Se ci sono conflitti a causa di due aggiornamenti che si accavallano, solitamente la cosa viene notata dal secondo che esegue commit
- Significa che dal suo ultimo update, qualcun altro ha eseguito un commit
- Il caso viene risolto localmente con un nuovo update (ed eventuali adattamenti) prima del commit

## Conflitti (2)

- Se il conflitto non si manifesta durante il commit (perché non sono stati toccati gli stessi file), ma esiste comunque una inconsistenza, questa viene scoperta durante il build di integrazione
- In entrambi i casi il problema viene scoperto velocemente
- La cosa più urgente da fare in questi casi è riparare il build

## Conflitti (3)

- In un ambiente di integrazione continua non si dovrebbe mai lasciare un “failed build” troppo a lungo
- Un buon team dovrebbe avere più di un build corretto al giorno
- Ognuno può quindi sviluppare a partire dall’ultima versione del build, mantenendo quindi il delta tra sviluppo e successiva integrazione il più piccolo possibile

## Elementi di CI

- Quanto visto nell’esempio precedente dimostra CI (continuous integration) nel lavoro di tutti i giorni
- Vediamo quali sono gli elementi essenziali affinché tutto questo possa avvenire in modo naturale e senza grossi problemi

## Elemento 1: Un solo repository

- I progetto software richiedono la gestione di parecchi file: utilizzare un sistema di versioning control
- Fare in modo che sia accessibile a tutti, anche da remoto
- Nel repository dovrebbe esserci tutto quanto server per eseguire il build, inclusi script, property file, librerie, ecc.
- Regola: dev'essere possibile eseguire un check out su una macchina "verGINE" ed poter subito eseguire un build

## Elemento 2: Build automatico (1)

- Build significa trasformare i sorgenti in un sistema funzionante
- Questo può essere un processo complicato che include compilazione, spostamento di file, caricamento di uno schema di DB, ecc.
- Tutto questo può essere automatizzato ed è buona cosa che lo sia, perché fa risparmiare parecchio tempo

## Elemento 2: Build automatico (2)

- Ambienti di build automatico ne esistono parecchi: Make in Unix, Ant, Maven, Gradle nel mondo Java, Nant o MSBuild per .NET, ecc.
- A dipendenza delle necessità si deve poter creare build in modo condizionale, avendo a disposizione diversi target (build con codice di test integrato, con verifiche diverse, ecc.)
- Il build può essere creato via IDE, ma dev'essere in ogni caso possibile creare un master sul server, senza IDE

## Elemento 3: Build self-testing

- Build non significa solo compilazione e link, un programma compilato correttamente può avere errori di esecuzione
- Il modo migliore per verificare il funzionamento è inserire la chiamata ai test nel processo di build
- Se un test non passa, il build deve fallire

## **Elemento 4: Si integra ogni giorno (1)**

- L'integrazione è anche un mezzo per comunicare agli altri sviluppatori quali modifiche abbiamo apportato al codice
- Integrando regolarmente si scoprono prima eventuali conflitti di sincronizzazione tra sviluppatori e si possono correggere velocemente
- Conflitti che rimangono irrisolti per giorni o settimane, sono difficili da riparare

## **Elemento 4: Si integra ogni giorno (2)**

- Regola: ogni sviluppatore dovrebbe eseguire un commit almeno una volta al giorno
- Commit frequenti (anche più volte al giorno) incoraggiano lo sviluppatore a suddividere il suo lavoro in fasi di alcune ore l'una. Questo permette di “sentire” il progredire del progetto

## Elemento 5: Mantenere il build veloce (1)

- Un elemento essenziale dell'integrazione continua è il feedback veloce
- Le “guidelines” di eXtreme Programming parlano di un massimo di 10 minuti
- Molto spesso se c’è un problema a mantenere il build a 10 minuti, questo è provocato dai test, soprattutto quelli che fanno uso di servizi esterni, come DB

## Elemento 5: Mantenere il build veloce (2)

- In caso di build troppo lunghi, si deve fare in modo di organizzare il processo a tappe (*staged build*, o *build pipeline*)
- Si parte da un “commit build” che deve durare al massimo 10 minuti, poi si possono organizzare passaggi successivi.
- Il commit build viene usato come punto di riferimento per il ciclo di integrazione continua

## Elemento 5: Mantenere il build veloce (3)

- **Esempio: two stage build**

- Il commit build si occupa della compilazione e dell'esecuzione dei test più essenziali, utilizzando oggetti "Mock" per i test sul DB
- Una volta eseguito questo (nei 10 minuti massimi) esiste un build non affidabile al 100%, ma sufficientemente sicuro da poter permettere agli sviluppatori di basare le proprie modifiche su questo ultimo build
- La seconda fase prevede l'utilizzo di tutti i test. A questo punto può anche durare alcune ore
- Se nella seconda fase si riscontrano problemi, si cerca di creare nuovi test per la prima fase che permettano di scoprire in anticipo quanto si è trovato di problematico

## Elemento 6: clonare l'ambiente di produzione (1)

- È importante poter eseguire tutti i test in un ambiente il più possibile simile a quello di produzione
- Ogni differenza può essere motivo di errore in produzione
- Clonare significa avere le stesse versioni del software, del sistema operativo, del DB, stesse librerie, stesso HW
- Ci sono dei limiti (HW troppo caro), ma spesso i vantaggi li superano ampiamente

## **Elemento 6: clonare l'ambiente di produzione (2)**

- Quando l'applicazione deve poter andare in produzione su ambienti diversi (esempio: applicazione desktop) è quasi impossibile provare tutte le combinazioni
- In questi casi possono però aiutare gli ambienti virtuali (VMWare, Parallels, ecc.)

## **Elemento 7: Rendere accessibile il build (1)**

- Uno dei maggiori problemi nello sviluppo del software è sapere se si stanno sviluppando le funzionalità realmente desiderate dal committente
- Le persone trovano più semplice vedere qualcosa di incompleto, ma che permetta di capire meglio cosa si desidera e come esprimere
- I processi agili si basano su questo e traggono vantaggio da questo tipico comportamento umano

## **Elemento 7: Rendere accessibile il build (2)**

- In queste situazioni è importante che ogni sviluppatore abbia facile accesso all'ultimo build, o al build della sua ultima iterazione
- Lo scopo è quello di poterlo usare per dimostrazioni, test, o anche unicamente per verificare cosa è cambiato negli ultimi giorni
- Il repository deve essere organizzato in modo che ci sia una chiara sequenza storica dei build

## **Elemento 8: Visione dello stato (1)**

- CI serve anche alla comunicazione, perciò è importante che ognuno possa verificare lo stato del sistema e i cambiamenti effettuati
- I programmi di CI hanno un'interfaccia Web che permette di accedere alla storia dei cambiamenti, sapere chi ha fatto le modifiche, ecc.
- Il vantaggio di un'interfaccia Web consiste nel fatto che anche sviluppatori localizzati altrove hanno facile accesso alle informazioni

## Elemento 8: Visione dello stato (2)

- Interfaccia Web di TeamCity

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- = **CobraWunelli Maintenance Build Configuration**  
#build.563-2 Tests passed: 159 | No artifacts Changes (1) 10 Feb 17:56 (7m:37s)  
Idle Run
- = **CobraWunelli Trunk Build Configuration**  
#build.680 Tests passed: 191 | No artifacts Changes (1) 25 Feb 15:33 (12m:09s)  
Idle Pending (1) Run

Sandro Pedrazzini Integrazione continua 37

## Elemento 9: Deployment automatico

- Un elemento di integrazione continua è il deployment automatico
- Questo può essere utile soprattutto se si hanno diversi deployment in ambienti diversi: automatizzare significa evitare facili fonti di errore
- Se è compresa la funzionalità di deployment in produzione, una capacità interessante è quella del rollback automatico alla versione precedente: questo permette di eliminare la “tensione” tipica del deployment

## Benefici della CI (1)

- Minor rischio
  - Ricordo di progetti senza integrazione continua: progetto terminato, ma incognita dell'integrazione
  - Difficile prevedere il tempo necessario di un'integrazione prevista solo alla fine del progetto
  - CI permette di sapere in ogni momento a che punto si è con il progetto e con gli errori

## Benefici della CI (2)

- Errori
  - CI non ci permette di eliminare gli errori, ma ci rende più semplice il compito di trovarli
  - Quando si introduce un bug nel sistema, se lo si scopre in fretta, diventa semplice anche toglierlo
  - Inoltre l'errore può essere solo introdotto in poche ore di lavoro dall'ultimo build stabile
  - Si eliminano i bug accumulati nel tempo

## Benefici della CI (3)

- Deployment più frequente
  - Grossa barriera che CI permette di eliminare
  - Permette di mostrare più rapidamente nuove features all'utente finale e ottenere di conseguenza un feedback più veloce
  - Attraverso feedback più veloce, utente e sviluppatore migliorano il loro rapporto di collaborazione
  - Si accorcia la distanza che esiste tipicamente tra sviluppatore e utente, decisiva per uno sviluppo software di successo

## Integrazione con le altre pratiche (1)

- L'utilizzo di CI si integra bene con altre pratiche di progettazione e sviluppo trattate
  - Test di unità
  - Utilizzo di standard nel codice
  - Refactoring
  - Cicli di sviluppo corti
  - Appartenenza comune del codice

## Integrazione con le altre pratiche (2)

- **Test di unità**

- Chi sviluppa, dovrebbe aggiungere codice di test al proprio codice (unit testing)
- Il test dovrebbe essere richiamato dopo ogni cambiamento
- Con CI il test viene richiamato automaticamente ad ogni build, quindi ad ogni modifica nel repository (regression test)

## Integrazione con le altre pratiche (3)

- **Utilizzo di standard**

- In ogni progetto si definiscono delle *guidelines* da seguire, in modo che l'intero codice segua gli stessi "standard"
- Spesso il controllo dell'aderenza allo standard è un processo manuale
- Con CI si può inserire una serie di analisi statiche del codice nello script di build, in grado di generare un report

## Integrazione con le altre pratiche (4)

- **Refactoring**

- Refactoring significa adattare il codice e la sua struttura interna senza modificare la funzionalità
- Uno degli scopi è quello di facilitare la manutenzione del codice
- CI assiste lo sviluppatore permettendo la chiamata di tool di inspection del codice all'interno del build, eseguito ad ogni modifica del repository

## Integrazione con le altre pratiche (5)

- **Cicli brevi**

- Significa che gli utenti devono avere a disposizione l'ultima versione del software funzionante il più spesso possibile
- CI si adatta bene a questa pratica, perché si integra più volte al giorno e ogni integrazione genera virtualmente un nuovo release
- Quando un sistema di integrazione continua è installato, un nuovo release viene generato con il minimo degli sforzi

## Integrazione con le altre pratiche (6)

- **Appartenenza comune (*collective ownership*)**
  - Ogni sviluppatore può lavorare a qualsiasi parte del sistema
  - Questo impedisce che ci sia un solo sviluppatore con conoscenze specifiche di un determinato argomento
  - CI aiuta questa pratica assicurando aderenza agli standard e richiamando continuamente i test di regressione

## Ridurre i rischi (1)

- **CI aiuta nel mantenere i rischi sotto controllo, permettendo di scoprire i problemi appena questi si manifestano**
- **Con CI e le pratiche correlate si riesce a creare una rete di qualità, che ci permette di fornire software di qualità più velocemente**

## Ridurre i rischi (2)

- Consideriamo i seguenti rischi
  - Software non deployable
  - Difetti scoperti tardi
  - Mancanza di visibilità del progetto
  - Software di bassa qualità
- Non si tratta di rendere attenti verso questi rischi (sono tutti noti), ma di permetterne la gestione

## Ridurre i rischi (3)

- Software non “deployabile”, impossibile creare il build

- Riesco a creare il build solo sulla mia macchina

In questo caso è importante sottolineare che il build dev'essere creato in modo indipendente dalla macchina, IDE utilizzato, configurazione specifica, ecc.  
È importante avere un server di integrazione, che usi uno script di build (ant) indipendente.

- Sincronizzazione con il DB

I test devono poter girare utilizzando l'ultima versione dello schema di DB. Lo schema di DB e i suoi dati minimi indispensabili devono trovarsi nel repository.  
I test devono essere in grado di eseguire un “drop” del DB e poi ricrearlo nuovo.

## Ridurre i rischi (4)

- **Difetti scoperti tardi**

- **Regression test**

Sappiamo che dobbiamo avere suite di test nel nostro progetto.  
Basta aggiungere la chiamata di questi test nello script di build, in questo modo verranno eseguiti dal server di integrazione ad ogni modifica.

- **Test coverage**

Esistono tool per verificare la percentuale di copertura dei test.  
Questi tool possono essere associati al processo di CI.

## Ridurre i rischi (5)

- **Mancanza di visibilità**

- **Informazioni**

È necessario che ogni sviluppatore nel progetto sia informato su ogni singola modifica effettuata al progetto.  
Al processo di CI può essere associata una notifica via email sulle modifiche effettuate o sugli eventuali problemi.

- **Visualizzazione grafica della struttura**

Se si desidera avere a disposizione l'ultima versione grafica della struttura del software (UML class diagram), lo si può fare inserendo nel sistema di CI la generazione a partire dall'ultima versione (esempio: Doxygen).

## Ridurre i rischi (6)

- **Software di bassa qualità (1)**

- **Aderenza del codice a standard predefiniti**

L'aderenza agli standard viene spesso controllata manualmente. In realtà esistono tool come Checkstyle, PMD o anche Sonar, che permettono di fare delle verifiche statiche del codice a partire da regole. Questi possono essere integrati al sistema CI.

- **Aderenza all'architettura**

Anche a livello di architettura ci possono essere delle guideline da seguire, ad esempio, il codice del data layer che non accede al codice del business layer, ecc.

Anche queste possono essere controllate da tool (JDepend, NDepend, ecc.) integrabili nel processo di CI.

## Ridurre i rischi (7)

- **Software di bassa qualità (2)**

- **Duplicazione del codice**

Codice duplicato rende più difficili le modifiche e la manutenzione del progetto.

È difficile scoprire dove abbiamo duplicazione di codice all'interno del progetto. Più il progetto è grande, più sviluppatori lavorano e maggiore è la probabilità di avere codice in più parti che esegue la stessa funzionalità.

Anche in questo caso esistono tool (utility CPD del tool di analisi metriche PMD, Simian, ecc.) che possono essere integrati nel processo di CI

## Come introdurre CI (1)

- Tutte le pratiche viste fin qui servono a trarre il massimo beneficio da CI
- Per iniziare non serve però applicarle tutte assieme
- Il primo passo è senza dubbio quello dell'automazione del processo di build
  - Mettiamo il progetto sotto il controllo di un sistema di gestione dei sorgenti (esempio: Subversion)
  - Facciamo in modo che con un unico comando si possa creare un build

## Come introdurre CI (2)

- Il passo successivo potrebbe essere quello di introdurre suite di test nel build automatico
- Se si hanno già test di unità nel sistema, la cosa è molto semplice, basta richiamarli durante il build
- Inizialmente il build verrà fatto partire a mano. Poi, si potrà richiamarlo automaticamente, con uno script, una volta al giorno per il nightly build
- Come ultimo passaggio, possiamo installare un server di integrazione con un software di CI (esempio: TeamCity)

## Deepening

### Mocking

Prof. Sandro Pedrazzini

SUPSI, University of Applied Sciences of Southern Switzerland  
sandro.pedrazzini@supsi.ch

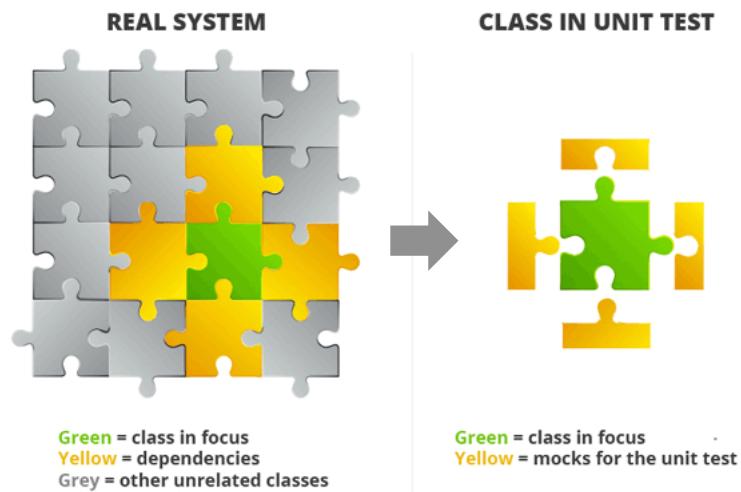
Canoo Engineering AG  
sandro.pedrazzini@canoo.com

## Mock Object

- **Objects simulating real objects' behaviour, used on controlled environments.**
- **They are used to verify the behaviour of other objects in a simplified way. If there were no mock, you would use the corresponding real objects.**
- **They are used during test. They encourage testing based on behaviour verification (in contrast to state verification).**

## Mock Object (2)

- Mock = fiction, imitation



## Content

- Motivation
- What is a mock
- State and behaviour verification
- Use with libraries: Mockito
- Terminology
- Testing styles

## Motivation

- Used to simulate complex objects, otherwise too expensive to create within unit tests.
- But:  
**Mocking also represents a different way/style of testing. There are differences**
  - in result verification (state vs. behaviour) and
  - in the way testing and design play together

## When mocking

- In such cases it is useful to provide and use mocking objects:
  - When the real object provides values that we cannot foresee (right because they are real, like date, time, temperature, etc.)
  - When the real object must get a state difficult to reproduce (example: connection error)
  - When the use of the real object would slow down the execution of the test (example: use of a DB)
  - When the real object does not exist yet
  - When the test would force the addition of methods and information only used for test purposes.

## Example 1: state verification

Wine Shop



- We have an *Order* object, used to post orders to an online wine shop.
- The wine shop holds an inventory of all wines and their bottles.

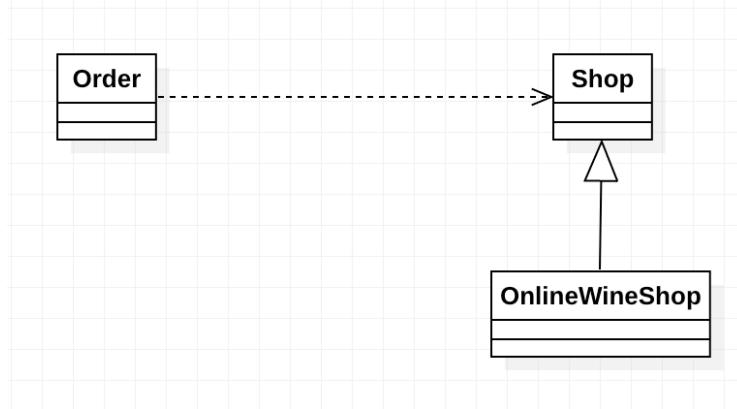
## Example 1: Scheme

- Dependency



## Example 1: Scheme (2)

- Use of Abstraction on Shop



## Example 1: Shop

```
public interface Shop {  
    void add(String productName, int quantity);  
    boolean consume(String productName, int requestedQuantity);  
    int getInventory(String productName);  
}
```

## Example 1: Order

```
public class Order {
    private String productName;
    private int quantity;
    private boolean isFilled;

    public Order(String productName, int quantity) {
        this.productName = productName;
        this.quantity = quantity;
        isFilled = false;
    }

    public void fill(Shop shop) {
        isFilled = shop.consume(productName, quantity);
    }

    ...
}
```

## Example 1: state verification

- When we ask an order to fill itself from the shop, there are two possible responses:
  - If there are enough bottles to fill the order, the order becomes filled and the shop's inventory will be reduced by the appropriate amount of bottles.
  - If the number of bottles is not enough to fill the order, the order is not filled.

## Example 1: state verification (2)

### Case 1: enough bottles

```
public class OrderTest {
    private static final String MERLOT = "Merlot";

    @Test
    public void testOrderIsFilledIfEnoughBottlesInShop() {
        Shop shop = new OnlineWineShop();
        shop.add(MERLOT, 20);
        Order order = new Order(MERLOT, 20);
        order.fill(shop);

        assertTrue(order.isFilled());
        assertEquals(0, shop.getInventory(MERLOT));
    }

    ...
}
```

## Example 1: state verification (3)

### Case 2: NOT enough bottles

```
public class OrderTest {
    private static final String MERLOT = "Merlot";

    ...

    @Test
    public void testOrderNotFilledIfNotEnoughBottlesInShop() {
        Shop shop = new OnlineWineShop();
        shop.add(MERLOT, 20);
        Order order = new Order(MERLOT, 21);
        order.fill(shop);

        assertFalse(order.isFilled());
        assertEquals(20, shop.getInventory(MERLOT));
    }
}
```

## Example 1: state verification (4)

- During each test we put together two objects: *Order* and *OnlineWineShop*.
- *Order* is the class that we are testing (SUT = system-under-test), but for *Order.fill()* to work we also need an instance of *OnlineWineShop* (collaborator).
- Reasons to need *OnlineWineShop*:
  - To get the test to work (since *Order.fill()* calls shop's methods).
  - For verification (since one of the results of *Order.fill()* is the potential change to the state of the shop).

## Example 1: state verification (5)

- State verification:

we determine whether the method worked correctly by examining the state of the SUT (Order object) and its collaborators (OnlineWineShop object).
- Mock objects enable a different approach to the verification phase

## Test with Mock Objects

- We take the same behaviour and use mock objects
- We use Mockito as Java mock library
- The mock library is used to replace components with objects with a predefined behaviour, only for test.

## Test with Mock Objects (2)

- A mock object is an object configured to return a specific output for a specific input, without performing a real action.
- Example:

```
OnlineWineShop shopMock =  
    Mockito.mock(OnlineWineShop.class);  
  
when(shopMock.consume(MERLOT, 20)).thenReturn(true);  
  
OR  
  
doReturn(true).when(shopMock).consume(MERLOT, 20);
```

## Example 1: behaviour verification

```
public class OrderTest {
    private static final String MERLOT = "Merlot";

    @Test
    public void testOrderIsFilledIfEnoughBottlesInShop() {
        Shop shopMock = Mockito.mock(Shop.class);
        when(shopMock.consume(MERLOT, 20)).thenReturn(true);

        Order order = new Order(MERLOT, 20);
        order.fill(shopMock);

        //verify state on order object
        assertTrue(order.isFilled());

        //verify behaviour on shop object (method is called)
        verify(shopMock).consume(MERLOT, 20);
    }

    ...
}
```

## Example 1: behaviour verification (2)

- **Setup phase**
  - The setup phase **is different**
  - **The SUT element is the same (Order).**
  - **The collaborator (Shop / OnlineWineShop) is NOT the same.**  
It is not an instance of OnlineWineShop, rather a mock of it (or of its interface Shop)
  - In the second phase of mock setup, **we set the expectations** on the mock object.

## Example 1: behaviour verification (3)

- **Execution and verification**

- The execution remains the same:

```
order.fill(shop);
```

- The verification has two aspects:

- **assert against the SUT** (as before) and
  - (new) **check that the mock was called** according to their expectations

## Example 1: behaviour verification (4)

- The **difference is how we verify** that the order object did the right thing in its interaction with the Wine shop
- With state verification we do this by asserts against the shop's state:

```
assertEquals(0, shop.getInventory(MERLOT));
```

- With behaviour verification we check to see if the order made the correct calls on the shop:

```
verify(shopMock).consume(MERLOT, 20);
```

## Example 1: behaviour verification (5)

- **Behaviour verification:**
  - First, tell the mock what we expect (setup)
  - Second, ask the mock to verify itself during verification
- Only the *Order* object is checked using asserts.
- If the method does not change the state of the SUT (*Order* object in this case), there is no need for asserts at all.

## Verification in Mockito: Method Calls

- We have verified that the method was called:

```
verify(shopMock).consume(MERLOT, 20);
```

- Mockito provides other kinds of verifications:

```
verify(shopMock, times(1)).consume(MERLOT, 20);
verify(shopMock, atLeastOnce()).consume(MERLOT, 20);
verify(shopMock, atLeast(1)).consume(MERLOT, 20);
verify(shopMock, atMost(1)).consume(MERLOT, 20);
verify(shopMock, never()).consume(MERLOT, 20);
```

## Verification in Mockito: Timeout

- Mockito verification also allows to specify a timeout for the mock methods execution.
- So, if we want to ensure that our `consume()` method is called within 100 milliseconds:

```
verify(shopMock, timeout(100)).consume(MERLOT, 20);
```

## Verification in Mockito: Void Method

- Let's consider that the call to:

```
order.fill(shop);
```

not only triggers an internal call to `shop.consume(...)` returning a boolean, but also a call to a further **void** shop method, called `shop.doNothing()`.

### Example

```
public void fill(Shop shop) {  
    shop.doNothing();  
    isFilled = shop.consume(...);  
    ...  
}
```

## Verification in Mockito: Void Methods (2)

- If we are interested if the void method is called, we do not need to set the *when* condition, we can simply verify the method call

```
@Test
public void testOrderWithVoid() {
    Shop shopMock = mock(Shop.class);

    Order order = new Order(MERLOT, 20);
    order.fill(shopMock);

    verify(shopMock).doNothing();
}
```

## Verification in Mockito: Void Methods (3)

- Only if we expect an exception from the void method, we need to specify a *do(...)* command

```
@Test(expected = RuntimeException.class)
public void testOrderWithVoidAndException() {
    Shop shopMock = mock(Shop.class);

    doThrow(new RuntimeException()).when(shopMock).doNothing();

    Order order = new Order(MERLOT, 20);
    order.fill(shopMock);
}
```

## Verification in Mockito: Call order

- Let's consider again that the call to:

```
order.fill(shop);
```

not only triggers an internal call to `shop.consume(...)`, but also a call to a further shop method, called `shop.doNothing()`, called **before** `consume()`

- Example**

```
public void fill(Shop shop) {
    shop.doNothing();
    isFilled = shop.consume(...);
    ...
}
```

## Verification in Mockito: Call Order (2)

- Mockito provides the class `InOrder`, **able to record the call order** and verify it:

```
@Test
public void testCallOrder() {
    Shop shopMock = mock(Shop.class);

    Order order = new Order(MERLOT, 21);
    order.fill(shopMock);

    //verify order in which methods are called
    InOrder inOrder = inOrder(shopMock);
    inOrder.verify(shopMock).doNothing();
    inOrder.verify(shopMock).consume(MERLOT, 21);
}
```

## Verification in Mockito: Call Order (3)

- If the order of calls is wrong (we expect in test `doNothing()` called after `consume()`), the test fails and following message is shown:

```
org.mockito.exceptionsverification.VerificationInOrderFailure:  
Verification in order failure  
  
Wanted but not invoked:  
onlineWineShop.doNothing();  
-> at ch.supsi.samples.mock.OrderWithMockTest.testCallOrder  
(OrderWithMockTest.java:53)  
  
Wanted anywhere AFTER following interaction:  
onlineWineShop.consume("Merlot", 21);  
-> at ch.supsi.samples.mock.Order.fill(Order.java:16)
```

## Verification in Mockito: any arguments

- In some situations the exact **values of the parameters do not care**.
- In case of order verification, we want to verify the order of the method calls, **we may not be interested** in the exact values of the passed parameters.
- For such cases Mockito provides the static methods of the `ArgumentMatchers` class.

### Examples:

- `anyString()`
- `anyInt()`
- `anyDouble()`
- `anyList()`
- `anyListOfClass(...)`
- ...

## Verification in Mockito: any arguments (2)

```
@Test
public void testCallOrder() {
    Shop shopMock = mock(Shop.class);

    Order order = new Order(MERLOT, 21);
    order.fill(shopMock);

    //verify order in which methods are called
    InOrder inOrder = inOrder(shopMock);
    inOrder.verify(shopMock).doNothing();
    inOrder.verify(shopMock).consume(anyString(), anyInt());
}
```

## Verification in Mockito: Exception

- Mockito allows its mocks to throw exceptions:

```
@Test (expected = RuntimeException.class)
public void testMockThrowException() {
    Shop shopMock = mock(Shop.class);
    when(shopMock.consume(MERLOT, 0))
        .thenThrow(new RuntimeException());

    Order order = new Order(MERLOT, 0);
    order.fill(shopMock);
}
```

## Mockito: Mock Annotation

- For a few mocks, creating every mock object is not a problem.

```
Shop shopMock = mock(Shop.class);
```

But, when there is a considerable number of them, it can be quite tedious to create all of them.

- Mockito provides creation through annotation
- The test class must run with *MockitoJUnitRunner*

## Mockito: Mock Annotation (2)

```
@RunWith(MockitoJUnitRunner.class)
public class OrderWithMockDITest {
    ...

    @Mock
    private Shop shopMock;

    @Test
    public void testOrderWithAnnotatedMock() {
        when(shopMock.consume(MERLOT, 20)).thenReturn(true);

        Order order = new Order(MERLOT, 20);
        order.fill(shopMock);

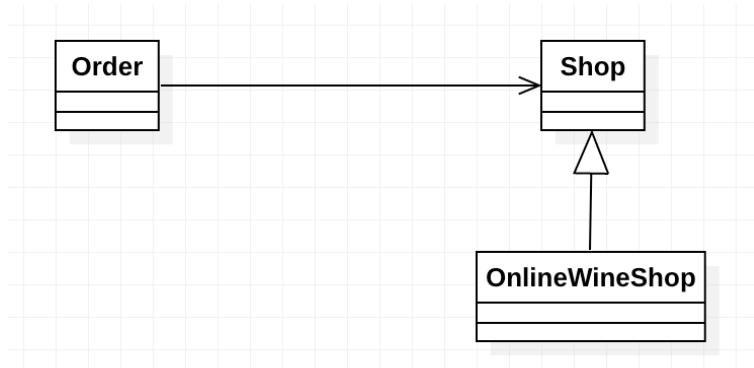
        //verify state
        assertTrue(order.isFilled());

        //verify that the method is called
        verify(shopMock).consume(MERLOT, 20);
    }
}
```

## Mockito: Mock Injection

- Mockito provides also mock injection as dependency of other objects (DI)
- Let's consider that the *Order* object has not only a dependency, but is really composed by the *Shop* object.
- Injecting the dependency means injecting the mock object into the *Order* one (SUT)

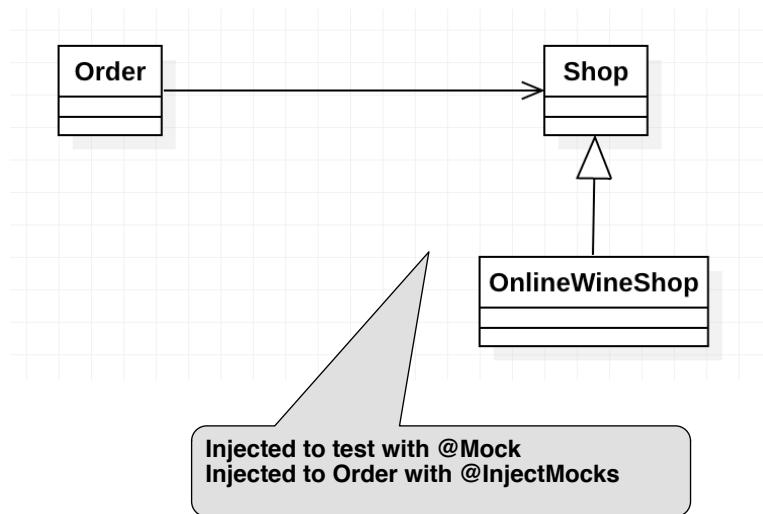
## Mockito: Mock Injection (2)



## Mockito: Mock Injection (3)

```
@RunWith(MockitoJUnitRunner.class)
public class OrderWithMockDITest {
    ...
    @Mock
    private Shop shopMock;
    @InjectMocks
    private Order order;      //with shop injected
    @Test
    public void testOrderWithDI() {
        when(shopMock.consume(MERLOT, 20)).thenReturn(true);
        order.fill();
        ...
    }
}
```

## Mockito: Mock Injection (4)



## Terminology

- In the two styles of testing we have seen, the first case uses a real *OnlineWineShop* object, whereas the second case uses a mock *OnlineWineShop* (or simply *Shop*).
- Using mocks is one way to avoid using a real Shop in the test, but **there are other forms of unreal objects used in testing.**
- The most generic term for all of them is **Test Double**:  
“any kind of object used in place of a real object for testing purposes” (Gerard Meszaros)

## Terminology (2)

- We distinguish five particular kinds of double:
  - **Dummy** objects are passed around but never actually used.  
Example: used to fill parameter lists.
  - **Fake** objects have working implementations, but usually take some shortcut which makes them not suitable for production.  
Example: in memory database.
  - **Stubs** provide answers to calls made during the test, usually not responding at all to anything outside what is foreseen for the test.  
Example: email service simulating the status change of a real email service, without sending mails.
  - **Mocks** are objects pre-programmed with expectations which form a specification of the calls they are expected to receive.
  - **Spies** are doubles that replicate real objects, eventually mocking them in some single parts.

## Terminology (3)

- Mocks actually **behave like other doubles** during the exercise phase, as they need to make the SUT believe it is talking with its real collaborators
- What distinguishes mocks, however, are the **setup** and the **verification phase**.
- Only mocks insist upon **behaviour verification**.

## Example 2: Email Service

- Many people only use test doubles if the real object is awkward to work with.
- A more common case for a test double would be if we decide that we want to send an email each time we fail to fill an order.
- **The problem:**  
we do not want to send an email message during testing.
- **Solution:**  
we create a test double of our email system.

## Example 2: Stub Solution

- We could write a simple stub of the Mail Service

```
public interface MailService {
    void send(Message message);
}

public class MailServiceStub implements MailService {
    private List<Message> messages = new ArrayList<>();

    public void send(Message message) {
        messages.add(message);
    }

    public int numberSent() {
        return messages.size();
    }
}
```

## Example 2: Stub Solution (2)

- With stubs we can simply use **state verification**

```
@Test
public void testOrderSendsMailWithStub () {
    Order order = new Order(MERLOT, 50);
    MailServiceStub mailerStub = new MailServiceStub();
    order.setMailer(mailerStub);

    order.fill(shop);

    assertEquals(1, mailer.numberSent());
}
```

## Example 2: Mock Solution

- The solution with MailService mock is quite different, above all in its **behaviour verification**.

```
@Test
public void testOrderSendsMailWithMock() {
    Order order = new Order(MERLOT, 50);
    MailService mailerMock = mock(MailService.class);
    order.setMailer(mailerMock);

    order.fill(shop);

    verify(mailerMock).send(any());
}
```

## Comparison

- In both cases we used a test double of the real mail service.
- Stub uses state verification, mock uses behaviour verification.**
- In order to use state verification on the stub, we need to provide some **extra method on the stub**, to help reading the changed state.

## Spy in Mockito

- A Spy is a **copy of a real object**.

```
public void simpleSpy() {
    List<String> list = new ArrayList<>();
    List<String> spyList = spy(list);

    assertEquals(0, spyList.size());

    spyList.add("one");
    spyList.add("two");

    verify(spyList).add("one");
    verify(spyList).add("two");
    assertEquals(2, spyList.size());
    assertEquals(0, list.size());
}
```

## Spy in Mockito (2)

- A Spy is a **copy of a real object, with mocked functionalities**.

```
public void mockedSpy() {
    List<String> list = new ArrayList<>();
    List<String> spyList = spy(list);

    assertEquals(0, spyList.size());

    spyList.add("one");
    spyList.add("two");

    assertEquals(2, spyList.size());

    //Mocked functionality
    doReturn(100).when(spyList).size();

    assertEquals(100, spyList.size());
}
```

## Mock vs. Spy in Mockito

- When Mockito creates a mock – it does so from a class, not from an actual instance. The mock simply creates an instance of the Class, entirely instrumented to track interactions with it.
- On the other hand, the spy will wrap an existing instance. It will still behave in the same way as the normal instance. The only difference is that it will also be instrumented to track all the interactions with it.

## Mock vs. Spy in Mockito (2)

- Mock of ArrayList

```
@Test
public void whenCreateMock_thenCreated() {
    List mockedList = mock(ArrayList.class);

    mockedList.add("one");

    verify(mockedList).add("one");

    assertEquals(0, mockedList.size());
}
```

## Mock vs. Spy in Mockito (3)

- Spy of ArrayList

```
@Test
public void whenCreateSpy_thenCreate() {
    List<String> spyList = spy(new ArrayList<>());
    spyList.add("one");
    verify(spyList).add("one");
    assertEquals(1, spyList.size());
}
```

## Classical vs Mockist Testing Style

- The classical testing style

is to use real objects if possible and a double if it is awkward to use the real thing.

Example:

In our case a classical TDDer would use a real shop and a double for the mail service.

## Classical vs Mockist Testing Style (2)

- **The mockist testing style**

is to always use a mock for any object with interesting behaviour outside SUT.

Example:

In our case a mockist would use mocks for the shop, as well as for the mail service.

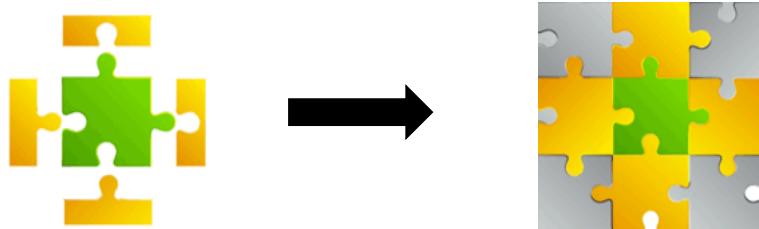
## Mockist Testing Style

- Use of “need-driven” development style: you begin developing a user story by writing your first test for the outside of your system.
- The system is your SUT, the rest are doubles.
- By thinking through the expectations upon the collaborators, you explore the interaction between the SUT and its neighbors, effectively designing the outbound interface of the SUT.



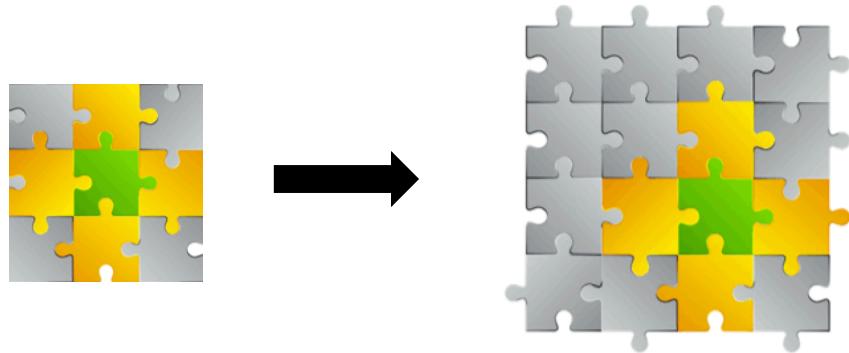
## Mockist Testing Style (2)

- Once you have your first test running, the expectations on the mocks provide a specification for the next step and a starting point for the tests.



## Mockist Testing Style (3)

- You turn each expectation into a test on a collaborator and repeat the process working your way into the system **one SUT at a time**.
- This style is also referred to as **outside-in**, which is a very descriptive name for it.



## Classic Testing Style

- Classic testing style (in particular classic TDD style) doesn't provide quite the same guidance.
- You can do a similar stepping approach, **using stubbed methods instead of mocks**.
- To do this, whenever you need something from a collaborator you **just hard-code exactly the response** the test requires to make the SUT work.
- Then once you're green with that you **replace the hard coded response with a proper code**.

## Classic Testing Style (2)

- But Classic testing style can do other things too.
- A common style is taking a feature and decide **what you need in the domain** for this feature to work.
- You **get the domain objects to do what you need** and once they are working you layer the UI on top.
- Doing this **you might never need to fake anything**.
- A lot of people like this because it focuses attention on the **domain model first**, which helps keep domain logic from leaking into the UI.

## Fixture Setup

- **With classic tests**, you have to create not just the SUT but also all the collaborators that the SUT needs in response to the test. While our example only had a couple of objects, real tests often involve a large amount of secondary objects. Usually these objects are created and torn down with each run of the tests.
- **With mockist tests**, however, you only need to create the SUT and mocks for its immediate neighbors. This can avoid some of the involved work in building up complex fixtures.

## Fixture Setup (2)

- In practice, **classic testers** tend to **reuse complex fixtures as much as possible**.
- In the simplest way you do this by putting fixture setup code into the unit test setup method. More complicated fixtures need to be used by several test classes, so in this case you create **special fixture generation classes** (object mothers).
- As a result both styles accuse the other of **creating too much work**.
  - Mockists say that creating the fixtures is a lot of effort,
  - Classicists say that this is reused and, on the other hand, with mockists you have to create mocks with every test.

## Test Isolation

### Dependencies

- If you introduce a bug to a system with **mockist testing**, it will usually cause only tests whose SUT contains the bug to fail.
- With the **classic approach**, however, any tests of client objects can also fail, which leads to failures where the buggy object is used as a collaborator in another object's test.
  - Usually the culprit is relatively easy to spot, however.
  - Moreover, one single change, can fix everything.

## Test Isolation (2)

### Granularity

- Since **classic tests** exercise multiple real objects, you often find a single test as the primary test for a **cluster of objects**, rather than just one. If that cluster spans many objects, then it can be much harder to find the real source of a bug.
- **Mockist tests** are less likely to suffer from this problem, because the convention is to mock out all objects beyond the primary.

## Test Isolation (3)

### Integration tests

- In essence **classic unit tests** are not just unit tests, but also mini-integration tests. As a result many people like the fact that client tests may catch errors that the main tests for an object may have missed, particularly probing areas where classes interact.
- **Mockist tests** lose that quality.

## Test Isolation (4)

### Coupling to the implementation

- A **classic test** only cares about the final state - not how that state was derived.
- **Mockist tests** are thus more coupled to the implementation of a method. With mockist testing, writing the test makes you think about the implementation of the behavior.
  - Classicists think that it's important to **only think about what happens from the external interface** and to leave all consideration of implementation until after you're done writing the test.
  - Coupling to the implementation also **interferes with refactoring**, since implementation changes are much more likely to break tests than with classic testing.

## Mockist / Classical Devide

- Often people learn a bit about the mock object frameworks, without fully understanding the **mockist/classical divide**.
- Whichever side of that divide you lean on, it's useful to understand this **difference in views**.
- You don't have to be a mockist, however, to **find the mock frameworks handy and appreciate their usefulness**, even if only in particular cases.

## Qualità nei processi di sviluppo

### Integrazione continua

### Integrazione continua

- Pratica dello sviluppo software
- Ogni membro del gruppo di sviluppo integra il proprio lavoro il più frequentemente possibile, solitamente almeno una volta al giorno
- Ogni integrazione viene verificata da un *build* automatico, che lancia i test e verifica eventuali problemi di integrazione

## Integrazione continua (2)

- Molti team di sviluppo trovano che una pratica del genere serve a ridurre al minimo i problemi di integrazione
- Permette al team di sviluppare software coeso in modo molto più rapido
- Aiuta a ridurre i rischi legati allo sviluppo e all'integrazione

## Motivazione

- In molti processi software l'integrazione richiede giorni, settimane o mesi
- Questo perché l'integrazione viene eseguita come ultimo passaggio prima di andare in produzione
- Per poter andare in produzione più frequentemente bisogna trasformare l'integrazione in qualcosa di automatico (integrazione vista come “non-event”)

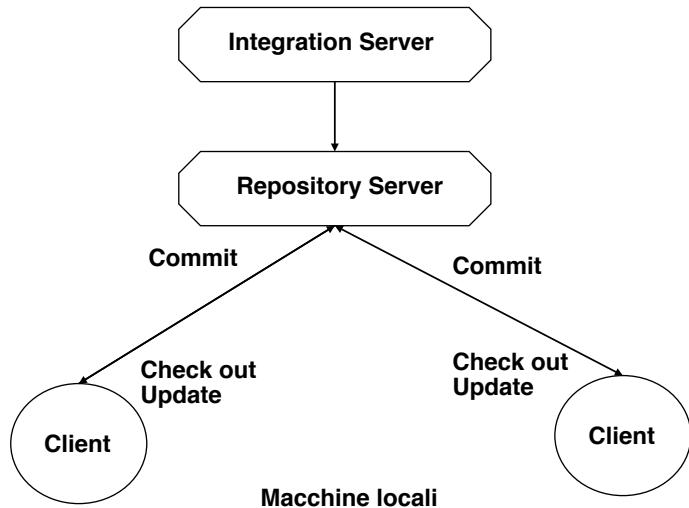
## Motivazione (2)

- Integrando regolarmente significa che ogni sviluppatore ad ogni iterazione aggiunge solo alcune ore di lavoro al progetto integrato
- Anche in questo caso (come nelle altre pratiche di XP) si tratta semplicemente di applicare in modo coerente alcune “buone abitudini”.
- Più frequentemente si integra e più l’integrazione diventa un “non-event”

## Tool

- Integrazione continua è prima di tutto una buona abitudine, perciò di base non necessitano tool particolari
- Ne esistono però alcuni (esempi: **CruiseControl**, **TeamCity**, **Jenkins**) che combinati a un sistema di gestione concorrente di sorgente (**CVS**, **Subversion**, **GIT**, ecc.) e a un sistema di build indipendente da ogni IDE (esempio: Ant, Maven, Gradle) permettono di attivare le verifiche di integrazione in modo automatico.

## Tool (2)

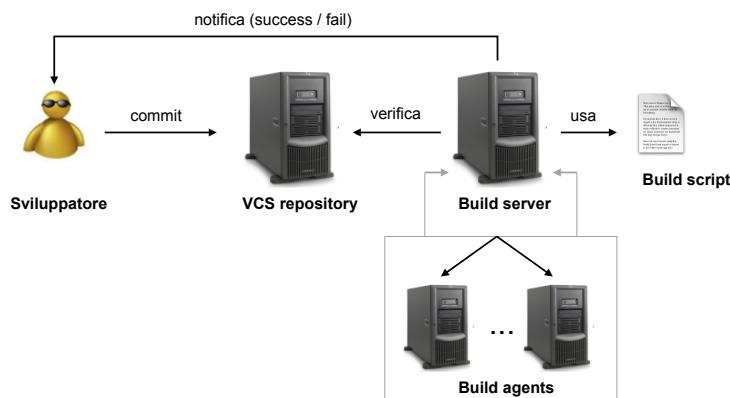


Sandro Pedrazzini

Integrazione continua

7

## Processo



Sandro Pedrazzini

Integrazione continua

8

## Tool (3)

- Esempio di build con Ant

➤ ant integrate

```
Buildfile: build.xml
clean:
all:
compile-src:
compile-tests:
integrate-db:
run-tests:
run-inspections:
delivery:
deploy:
BUILD SUCCESSFUL
Total time: 6 minutes 15 seconds
```

## Tool (4)

- Esempio di commit con Subversion (via linea di comando)

➤ svn commit -m "Aggiunta verifica della connessione"

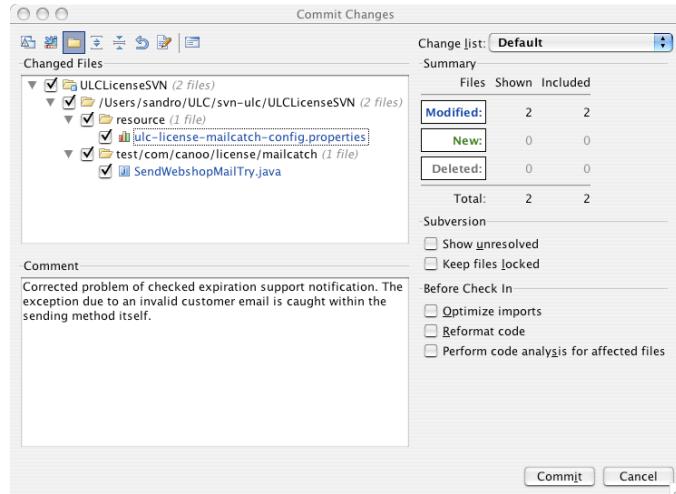
```
Sending src/com/canoo/db/dao/DBServicesImpl.java
Transmitting file data .


```

```
Committed revision 125
```

## Tool (5)

- Esempio di commit integrato in IDE



Sandro Pedrazzini

Integrazione continua

11

## Tool (6)

- Confronto fra versioni

The screenshot shows a code comparison tool displaying the differences between two versions of the 'CheckExpiringSupport.java' file. The tool highlights changes in green, deletions in red, and insertions in blue. The code itself is a Java class with various methods and logic for handling license expiration notifications.

Sandro Pedrazzini

Integrazione continua

12

## Tool (8)

- Mail ricevuta da Cruise Control dopo il build

### BUILD COMPLETE - build.180

Date of build: 01/07/2008 16:21:50  
Time to build: 10 seconds  
Last changed: 01/07/2008 16:16:51  
Last log entry: Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Modifications since last successful build: (6)

modified sandro /ULCLicense/trunk/src/com/canoo/license/base/data/transaction/TransactionContext.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
added sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification-3.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/src/com/canoo/license/base/support/CheckExpiringSupport.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/build.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicenseTemplates/trunk/ULCBase/ulc-support-additions.sql	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Sandro Pedrazzini

Integrazione continua

13

## Tool (9)

### Progetti in TeamCity

Welcome, sandro Logout

Projects My Changes Agents (1) Build Queue (0) Administration My Settings & Tools Configure Visible Projects

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- **CobraWunelli Maintenance Build Configuration** Idle Run
- #build.563-2 Tests passed: 159
- No artifacts Changes (1) 10 Feb 17:56 (7m:37s)
- **CobraWunelli Trunk Build Configuration** Idle Pending (1) Run
- #build.680 Tests passed: 191
- No artifacts Changes (1) 25 Feb 15:33 (12m:09s)

Sandro Pedrazzini

Integrazione continua

14

## Tool (10)

- History di TeamCity

The screenshot shows the TeamCity web interface for the project 'CobraWunelli'. The top navigation bar includes 'Projects', 'My Changes', 'Agents (1)', 'Build Queue (0)', 'Administration', 'My Settings & Tools', and a search bar. The main content area displays the 'History' tab of the build configuration 'CobraWunelli Trunk Build Configuration'. It shows a table of recent builds with columns for '#', 'Results', 'Artifacts', 'Changes', 'Started', 'Duration', 'Agent', 'Tags', and a 'Pin' icon. The table lists builds from #build.680 to #build.666. Build #680 is green (Tests passed: 191). Builds #678, #677, #676, #672, #671, #670, and #669 are green (Tests passed: 185, 172, 172, 175, 175, 175, 175). Build #668 is red (Tests failed: 1 (1 new), passed: 171). Build #666 is red ([Execution timeout] Tests passed: 171). The status bar at the bottom indicates 'Integrazione continua' and the page number '15'.

## Esempio (1)

- Supponiamo di voler aggiungere una nuova funzionalità ad un programma esistente

- Come primo passaggio devo scaricare la versione più aggiornata sulla mia macchina locale (check-out o update, nel caso avessi già una versione locale non aggiornata)
- Quando ho la versione sulla mia macchina, posso iniziare a fare le aggiunte desiderate
- Questo significa non solo aggiungere funzionalità, ma anche adattare vecchio codice, aggiungere e adattare test

## Esempio (2)

- Appena terminato con la nuova funzionalità e i vari test, posso creare il build sulla mia macchina locale
- Questo significa ricompilare tutto, creare e includere le varie librerie, eseguire i test
- Posso considerare di aver terminato il lavoro sulla macchina locale solo quando tutto funziona senza errori
- Ora che il build locale funziona, posso pensare di eseguire un “commit” sul repository comune

## Esempio (3)

- Non ho ancora finito: a questo punto si tratta di eseguire il build sulla macchina di integrazione
- Solo se anche questo build ha successo, si può affermare che i cambiamenti eseguiti fanno parte dell’applicazione
- Il build di integrazione può essere eseguito manualmente oppure automaticamente con tool come TeamCity, che, eseguendo un polling continuo sul repository per determinare se ci sono stati aggiornamenti, fa partire il build quando necessario

## Conflitti (1)

- Se ci sono conflitti a causa di due aggiornamenti che si accavallano, solitamente la cosa viene notata dal secondo che esegue commit
- Significa che dal suo ultimo update, qualcun altro ha eseguito un commit
- Il caso viene risolto localmente con un nuovo update (ed eventuali adattamenti) prima del commit

## Conflitti (2)

- Se il conflitto non si manifesta durante il commit (perché non sono stati toccati gli stessi file), ma esiste comunque una inconsistenza, questa viene scoperta durante il build di integrazione
- In entrambi i casi il problema viene scoperto velocemente
- La cosa più urgente da fare in questi casi è riparare il build

## Conflitti (3)

- In un ambiente di integrazione continua non si dovrebbe mai lasciare un “failed build” troppo a lungo
- Un buon team dovrebbe avere più di un build corretto al giorno
- Ognuno può quindi sviluppare a partire dall’ultima versione del build, mantenendo quindi il delta tra sviluppo e successiva integrazione il più piccolo possibile

## Elementi di CI

- Quanto visto nell’esempio precedente dimostra CI (continuous integration) nel lavoro di tutti i giorni
- Vediamo quali sono gli elementi essenziali affinché tutto questo possa avvenire in modo naturale e senza grossi problemi

## Elemento 1: Un solo repository

- I progetto software richiedono la gestione di parecchi file: utilizzare un sistema di versioning control
- Fare in modo che sia accessibile a tutti, anche da remoto
- Nel repository dovrebbe esserci tutto quanto server per eseguire il build, inclusi script, property file, librerie, ecc.
- Regola: dev'essere possibile eseguire un check out su una macchina "verGINE" ed poter subito eseguire un build

## Elemento 2: Build automatico (1)

- Build significa trasformare i sorgenti in un sistema funzionante
- Questo può essere un processo complicato che include compilazione, spostamento di file, caricamento di uno schema di DB, ecc.
- Tutto questo può essere automatizzato ed è buona cosa che lo sia, perché fa risparmiare parecchio tempo

## Elemento 2: Build automatico (2)

- Ambienti di build automatico ne esistono parecchi: Make in Unix, Ant, Maven, Gradle nel mondo Java, Nant o MSBuild per .NET, ecc.
- A dipendenza delle necessità si deve poter creare build in modo condizionale, avendo a disposizione diversi target (build con codice di test integrato, con verifiche diverse, ecc.)
- Il build può essere creato via IDE, ma dev'essere in ogni caso possibile creare un master sul server, senza IDE

## Elemento 3: Build self-testing

- Build non significa solo compilazione e link, un programma compilato correttamente può avere errori di esecuzione
- Il modo migliore per verificare il funzionamento è inserire la chiamata ai test nel processo di build
- Se un test non passa, il build deve fallire

## **Elemento 4: Si integra ogni giorno (1)**

- L'integrazione è anche un mezzo per comunicare agli altri sviluppatori quali modifiche abbiamo apportato al codice
- Integrando regolarmente si scoprono prima eventuali conflitti di sincronizzazione tra sviluppatori e si possono correggere velocemente
- Conflitti che rimangono irrisolti per giorni o settimane, sono difficili da riparare

## **Elemento 4: Si integra ogni giorno (2)**

- Regola: ogni sviluppatore dovrebbe eseguire un commit almeno una volta al giorno
- Commit frequenti (anche più volte al giorno) incoraggiano lo sviluppatore a suddividere il suo lavoro in fasi di alcune ore l'una. Questo permette di “sentire” il progredire del progetto

## Elemento 5: Mantenere il build veloce (1)

- Un elemento essenziale dell'integrazione continua è il feedback veloce
- Le “guidelines” di eXtreme Programming parlano di un massimo di 10 minuti
- Molto spesso se c’è un problema a mantenere il build a 10 minuti, questo è provocato dai test, soprattutto quelli che fanno uso di servizi esterni, come DB

## Elemento 5: Mantenere il build veloce (2)

- In caso di build troppo lunghi, si deve fare in modo di organizzare il processo a tappe (*staged build*, o *build pipeline*)
- Si parte da un “commit build” che deve durare al massimo 10 minuti, poi si possono organizzare passaggi successivi.
- Il commit build viene usato come punto di riferimento per il ciclo di integrazione continua

## Elemento 5: Mantenere il build veloce (3)

- **Esempio: two stage build**

- Il commit build si occupa della compilazione e dell'esecuzione dei test più essenziali, utilizzando oggetti "Mock" per i test sul DB
- Una volta eseguito questo (nei 10 minuti massimi) esiste un build non affidabile al 100%, ma sufficientemente sicuro da poter permettere agli sviluppatori di basare le proprie modifiche su questo ultimo build
- La seconda fase prevede l'utilizzo di tutti i test. A questo punto può anche durare alcune ore
- Se nella seconda fase si riscontrano problemi, si cerca di creare nuovi test per la prima fase che permettano di scoprire in anticipo quanto si è trovato di problematico

## Elemento 6: clonare l'ambiente di produzione (1)

- È importante poter eseguire tutti i test in un ambiente il più possibile simile a quello di produzione
- Ogni differenza può essere motivo di errore in produzione
- Clonare significa avere le stesse versioni del software, del sistema operativo, del DB, stesse librerie, stesso HW
- Ci sono dei limiti (HW troppo caro), ma spesso i vantaggi li superano ampiamente

## **Elemento 6: clonare l'ambiente di produzione (2)**

- Quando l'applicazione deve poter andare in produzione su ambienti diversi (esempio: applicazione desktop) è quasi impossibile provare tutte le combinazioni
- In questi casi possono però aiutare gli ambienti virtuali (VMWare, Parallels, ecc.)

## **Elemento 7: Rendere accessibile il build (1)**

- Uno dei maggiori problemi nello sviluppo del software è sapere se si stanno sviluppando le funzionalità realmente desiderate dal committente
- Le persone trovano più semplice vedere qualcosa di incompleto, ma che permetta di capire meglio cosa si desidera e come esprimere
- I processi agili si basano su questo e traggono vantaggio da questo tipico comportamento umano

## **Elemento 7: Rendere accessibile il build (2)**

- In queste situazioni è importante che ogni sviluppatore abbia facile accesso all'ultimo build, o al build della sua ultima iterazione
- Lo scopo è quello di poterlo usare per dimostrazioni, test, o anche unicamente per verificare cosa è cambiato negli ultimi giorni
- Il repository deve essere organizzato in modo che ci sia una chiara sequenza storica dei build

## **Elemento 8: Visione dello stato (1)**

- CI serve anche alla comunicazione, perciò è importante che ognuno possa verificare lo stato del sistema e i cambiamenti effettuati
- I programmi di CI hanno un'interfaccia Web che permette di accedere alla storia dei cambiamenti, sapere chi ha fatto le modifiche, ecc.
- Il vantaggio di un'interfaccia Web consiste nel fatto che anche sviluppatori localizzati altrove hanno facile accesso alle informazioni

## Elemento 8: Visione dello stato (2)

- Interfaccia Web di TeamCity

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- = **CobraWunelli Maintenance Build Configuration**  
#build.563-2 Tests passed: 159 | No artifacts Changes (1) 10 Feb 17:56 (7m:37s)  
Idle Run
- = **CobraWunelli Trunk Build Configuration**  
#build.680 Tests passed: 191 | No artifacts Changes (1) 25 Feb 15:33 (12m:09s)  
Idle Pending (1) Run

Sandro Pedrazzini Integrazione continua 37

## Elemento 9: Deployment automatico

- Un elemento di integrazione continua è il deployment automatico
- Questo può essere utile soprattutto se si hanno diversi deployment in ambienti diversi: automatizzare significa evitare facili fonti di errore
- Se è compresa la funzionalità di deployment in produzione, una capacità interessante è quella del rollback automatico alla versione precedente: questo permette di eliminare la “tensione” tipica del deployment

## Benefici della CI (1)

- Minor rischio
  - Ricordo di progetti senza integrazione continua: progetto terminato, ma incognita dell'integrazione
  - Difficile prevedere il tempo necessario di un'integrazione prevista solo alla fine del progetto
  - CI permette di sapere in ogni momento a che punto si è con il progetto e con gli errori

## Benefici della CI (2)

- Errori
  - CI non ci permette di eliminare gli errori, ma ci rende più semplice il compito di trovarli
  - Quando si introduce un bug nel sistema, se lo si scopre in fretta, diventa semplice anche toglierlo
  - Inoltre l'errore può essere solo introdotto in poche ore di lavoro dall'ultimo build stabile
  - Si eliminano i bug accumulati nel tempo

## Benefici della CI (3)

- Deployment più frequente
  - Grossa barriera che CI permette di eliminare
  - Permette di mostrare più rapidamente nuove features all'utente finale e ottenere di conseguenza un feedback più veloce
  - Attraverso feedback più veloce, utente e sviluppatore migliorano il loro rapporto di collaborazione
  - Si accorcia la distanza che esiste tipicamente tra sviluppatore e utente, decisiva per uno sviluppo software di successo

## Integrazione con le altre pratiche (1)

- L'utilizzo di CI si integra bene con altre pratiche di progettazione e sviluppo trattate
  - Test di unità
  - Utilizzo di standard nel codice
  - Refactoring
  - Cicli di sviluppo corti
  - Appartenenza comune del codice

## Integrazione con le altre pratiche (2)

- **Test di unità**

- Chi sviluppa, dovrebbe aggiungere codice di test al proprio codice (unit testing)
- Il test dovrebbe essere richiamato dopo ogni cambiamento
- Con CI il test viene richiamato automaticamente ad ogni build, quindi ad ogni modifica nel repository (regression test)

## Integrazione con le altre pratiche (3)

- **Utilizzo di standard**

- In ogni progetto si definiscono delle *guidelines* da seguire, in modo che l'intero codice segua gli stessi "standard"
- Spesso il controllo dell'aderenza allo standard è un processo manuale
- Con CI si può inserire una serie di analisi statiche del codice nello script di build, in grado di generare un report

## Integrazione con le altre pratiche (4)

- **Refactoring**

- Refactoring significa adattare il codice e la sua struttura interna senza modificare la funzionalità
- Uno degli scopi è quello di facilitare la manutenzione del codice
- CI assiste lo sviluppatore permettendo la chiamata di tool di inspection del codice all'interno del build, eseguito ad ogni modifica del repository

## Integrazione con le altre pratiche (5)

- **Cicli brevi**

- Significa che gli utenti devono avere a disposizione l'ultima versione del software funzionante il più spesso possibile
- CI si adatta bene a questa pratica, perché si integra più volte al giorno e ogni integrazione genera virtualmente un nuovo release
- Quando un sistema di integrazione continua è installato, un nuovo release viene generato con il minimo degli sforzi

## Integrazione con le altre pratiche (6)

- **Appartenenza comune (*collective ownership*)**
  - Ogni sviluppatore può lavorare a qualsiasi parte del sistema
  - Questo impedisce che ci sia un solo sviluppatore con conoscenze specifiche di un determinato argomento
  - CI aiuta questa pratica assicurando aderenza agli standard e richiamando continuamente i test di regressione

## Ridurre i rischi (1)

- **CI aiuta nel mantenere i rischi sotto controllo, permettendo di scoprire i problemi appena questi si manifestano**
- **Con CI e le pratiche correlate si riesce a creare una rete di qualità, che ci permette di fornire software di qualità più velocemente**

## Ridurre i rischi (2)

- Consideriamo i seguenti rischi
  - Software non deployable
  - Difetti scoperti tardi
  - Mancanza di visibilità del progetto
  - Software di bassa qualità
- Non si tratta di rendere attenti verso questi rischi (sono tutti noti), ma di permetterne la gestione

## Ridurre i rischi (3)

- Software non “deployabile”, impossibile creare il build

- Riesco a creare il build solo sulla mia macchina

In questo caso è importante sottolineare che il build dev'essere creato in modo indipendente dalla macchina, IDE utilizzato, configurazione specifica, ecc.  
È importante avere un server di integrazione, che usi uno script di build (ant) indipendente.

- Sincronizzazione con il DB

I test devono poter girare utilizzando l'ultima versione dello schema di DB. Lo schema di DB e i suoi dati minimi indispensabili devono trovarsi nel repository.  
I test devono essere in grado di eseguire un “drop” del DB e poi ricrearlo nuovo.

## Ridurre i rischi (4)

- **Difetti scoperti tardi**

- **Regression test**

Sappiamo che dobbiamo avere suite di test nel nostro progetto.  
Basta aggiungere la chiamata di questi test nello script di build, in questo modo verranno eseguiti dal server di integrazione ad ogni modifica.

- **Test coverage**

Esistono tool per verificare la percentuale di copertura dei test.  
Questi tool possono essere associati al processo di CI.

## Ridurre i rischi (5)

- **Mancanza di visibilità**

- **Informazioni**

È necessario che ogni sviluppatore nel progetto sia informato su ogni singola modifica effettuata al progetto.  
Al processo di CI può essere associata una notifica via email sulle modifiche effettuate o sugli eventuali problemi.

- **Visualizzazione grafica della struttura**

Se si desidera avere a disposizione l'ultima versione grafica della struttura del software (UML class diagram), lo si può fare inserendo nel sistema di CI la generazione a partire dall'ultima versione (esempio: Doxygen).

## Ridurre i rischi (6)

- **Software di bassa qualità (1)**

- **Aderenza del codice a standard predefiniti**

L'aderenza agli standard viene spesso controllata manualmente. In realtà esistono tool come Checkstyle, PMD o anche Sonar, che permettono di fare delle verifiche statiche del codice a partire da regole. Questi possono essere integrati al sistema CI.

- **Aderenza all'architettura**

Anche a livello di architettura ci possono essere delle guideline da seguire, ad esempio, il codice del data layer che non accede al codice del business layer, ecc.

Anche queste possono essere controllate da tool (JDepend, NDepend, ecc.) integrabili nel processo di CI.

## Ridurre i rischi (7)

- **Software di bassa qualità (2)**

- **Duplicazione del codice**

Codice duplicato rende più difficili le modifiche e la manutenzione del progetto.

È difficile scoprire dove abbiamo duplicazione di codice all'interno del progetto. Più il progetto è grande, più sviluppatori lavorano e maggiore è la probabilità di avere codice in più parti che esegue la stessa funzionalità.

Anche in questo caso esistono tool (utility CPD del tool di analisi metriche PMD, Simian, ecc.) che possono essere integrati nel processo di CI

## Come introdurre CI (1)

- Tutte le pratiche viste fin qui servono a trarre il massimo beneficio da CI
- Per iniziare non serve però applicarle tutte assieme
- Il primo passo è senza dubbio quello dell'automazione del processo di build
  - Mettiamo il progetto sotto il controllo di un sistema di gestione dei sorgenti (esempio: Subversion)
  - Facciamo in modo che con un unico comando si possa creare un build

## Come introdurre CI (2)

- Il passo successivo potrebbe essere quello di introdurre suite di test nel build automatico
- Se si hanno già test di unità nel sistema, la cosa è molto semplice, basta richiamarli durante il build
- Inizialmente il build verrà fatto partire a mano. Poi, si potrà richiamarlo automaticamente, con uno script, una volta al giorno per il nightly build
- Come ultimo passaggio, possiamo installare un server di integrazione con un software di CI (esempio: TeamCity)

## Qualità nei processi di sviluppo

### Integrazione continua

### Integrazione continua

- Pratica dello sviluppo software
- Ogni membro del gruppo di sviluppo integra il proprio lavoro il più frequentemente possibile, solitamente almeno una volta al giorno
- Ogni integrazione viene verificata da un *build* automatico, che lancia i test e verifica eventuali problemi di integrazione

## Integrazione continua (2)

- Molti team di sviluppo trovano che una pratica del genere serve a ridurre al minimo i problemi di integrazione
- Permette al team di sviluppare software coeso in modo molto più rapido
- Aiuta a ridurre i rischi legati allo sviluppo e all'integrazione

## Motivazione

- In molti processi software l'integrazione richiede giorni, settimane o mesi
- Questo perché l'integrazione viene eseguita come ultimo passaggio prima di andare in produzione
- Per poter andare in produzione più frequentemente bisogna trasformare l'integrazione in qualcosa di automatico (integrazione vista come “non-event”)

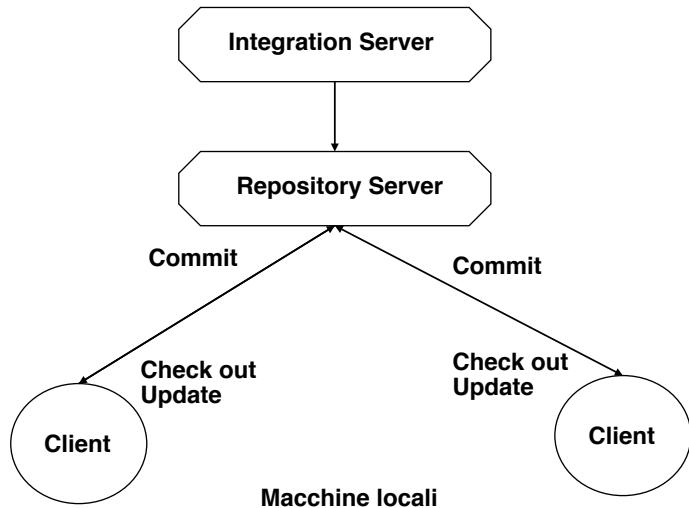
## Motivazione (2)

- Integrando regolarmente significa che ogni sviluppatore ad ogni iterazione aggiunge solo alcune ore di lavoro al progetto integrato
- Anche in questo caso (come nelle altre pratiche di XP) si tratta semplicemente di applicare in modo coerente alcune “buone abitudini”.
- Più frequentemente si integra e più l’integrazione diventa un “non-event”

## Tool

- Integrazione continua è prima di tutto una buona abitudine, perciò di base non necessitano tool particolari
- Ne esistono però alcuni (esempi: **CruiseControl**, **TeamCity**, **Jenkins**) che combinati a un sistema di gestione concorrente di sorgente (**CVS**, **Subversion**, **GIT**, ecc.) e a un sistema di build indipendente da ogni IDE (esempio: Ant, Maven, Gradle) permettono di attivare le verifiche di integrazione in modo automatico.

## Tool (2)

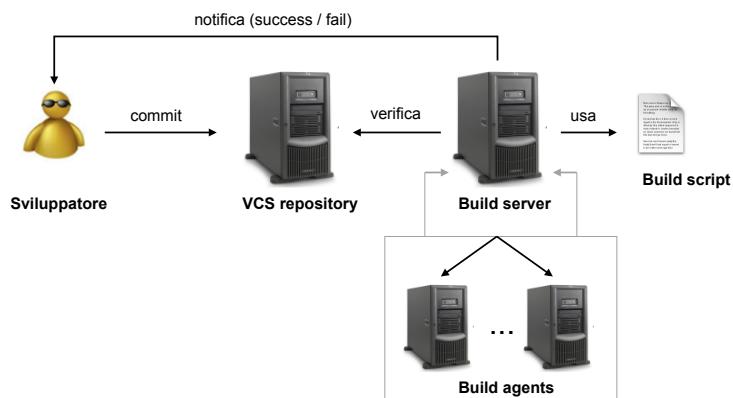


Sandro Pedrazzini

Integrazione continua

7

## Processo



Sandro Pedrazzini

Integrazione continua

8

## Tool (3)

- Esempio di build con Ant

➤ ant integrate

```
Buildfile: build.xml
clean:
all:
compile-src:
compile-tests:
integrate-db:
run-tests:
run-inspections:
delivery:
deploy:
BUILD SUCCESSFUL
Total time: 6 minutes 15 seconds
```

## Tool (4)

- Esempio di commit con Subversion (via linea di comando)

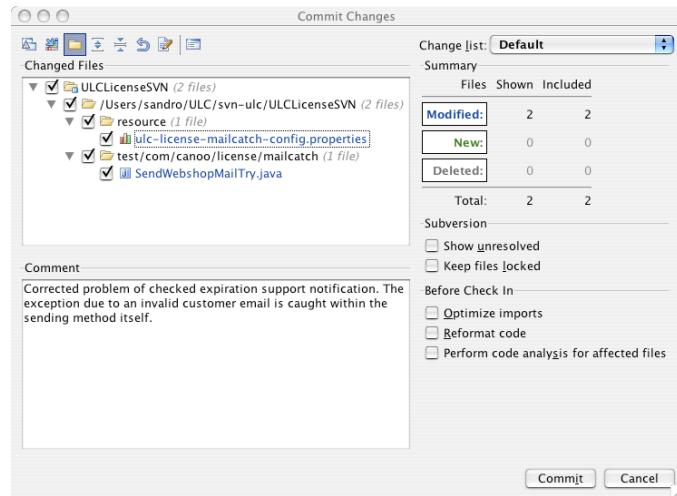
➤ svn commit -m "Aggiunta verifica della connessione"

```
Sending src/com/canoo/db/dao/DBServicesImpl.java
Transmitting file data .

Committed revision 125
```

## Tool (5)

- Esempio di commit integrato in IDE



Sandro Pedrazzini

Integrazione continua

11

## Tool (6)

- Confronto fra versioni

The screenshot shows a code comparison tool with two panes. The left pane displays the original code for 'CheckExpiringSupport.java' with line numbers 10351 down to 6. The right pane shows the updated code with various annotations: green highlights for inserted lines, red highlights for deleted lines, and blue highlights for changed lines. Annotations include messages like 'Message could not be sent, but the license will be marked as expired', 'Change body of catch statement use File / Settings', and 'Problem in ulc/checked'. The bottom status bar indicates there are 6 differences, 0 deleted, 10 changed, and 6 inserted.

Sandro Pedrazzini

Integrazione continua

12

## Tool (8)

- Mail ricevuta da Cruise Control dopo il build

### BUILD COMPLETE - build.180

Date of build: 01/07/2008 16:21:50  
Time to build: 10 seconds  
Last changed: 01/07/2008 16:16:51  
Last log entry: Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Modifications since last successful build: (6)

modified sandro /ULCLicense/trunk/src/com/canoo/license/base/data/transaction/TransactionContext.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
added sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification-3.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/test/com/canoo/license/mailcatch/notification.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/src/com/canoo/license/base/support/CheckExpiringSupport.java	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicense/trunk/build.xml	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.
modified sandro /ULCLicenseTemplates/trunk/ULCBase/ulc-support-additions.sql	Corrected problem of checked expiration support notification. The exception due to an invalid customer email is caught within the sending method itself.

Sandro Pedrazzini

Integrazione continua

13

## Tool (9)

### Progetti in TeamCity

Welcome, sandro Logout

Projects My Changes Agents (1) Build Queue (0) Administration My Settings & Tools Configure Visible Projects

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- **CobraWunelli Maintenance Build Configuration** Idle Run
- #build.563-2 Tests passed: 159
- No artifacts Changes (1) 10 Feb 17:56 (7m:37s)
- **CobraWunelli Trunk Build Configuration** Idle Pending (1) Run
- #build.680 Tests passed: 191
- No artifacts Changes (1) 25 Feb 15:33 (12m:09s)

Sandro Pedrazzini

Integrazione continua

14

## Tool (10)

- History di TeamCity

The screenshot shows the TeamCity web interface for the project 'CobraWunelli'. The top navigation bar includes 'Projects', 'My Changes', 'Agents (1)', 'Build Queue (0)', 'Administration', 'My Settings & Tools', and a search bar. The main content area displays the 'History' tab of the build configuration 'CobraWunelli Trunk Build Configuration'. It shows a table of recent builds with columns for '#', 'Results', 'Artifacts', 'Changes', 'Started', 'Duration', 'Agent', 'Tags', and a 'Pin' icon. The table lists builds from #build.680 to #build.666. Build #680 is green (Tests passed: 191). Builds #678, #677, #676, #672, #671, #670, and #669 are green (Tests passed: 185, 172, 172, 175, 175, 175, 175). Build #668 is red (Tests failed: 1 (1 new), passed: 171). Build #666 is red ([Execution timeout] Tests passed: 171). The status bar at the bottom indicates 'Integrazione continua' and the page number '15'.

## Esempio (1)

- Supponiamo di voler aggiungere una nuova funzionalità ad un programma esistente

- Come primo passaggio devo scaricare la versione più aggiornata sulla mia macchina locale (check-out o update, nel caso avessi già una versione locale non aggiornata)
- Quando ho la versione sulla mia macchina, posso iniziare a fare le aggiunte desiderate
- Questo significa non solo aggiungere funzionalità, ma anche adattare vecchio codice, aggiungere e adattare test

## Esempio (2)

- Appena terminato con la nuova funzionalità e i vari test, posso creare il build sulla mia macchina locale
- Questo significa ricompilare tutto, creare e includere le varie librerie, eseguire i test
- Posso considerare di aver terminato il lavoro sulla macchina locale solo quando tutto funziona senza errori
- Ora che il build locale funziona, posso pensare di eseguire un “commit” sul repository comune

## Esempio (3)

- Non ho ancora finito: a questo punto si tratta di eseguire il build sulla macchina di integrazione
- Solo se anche questo build ha successo, si può affermare che i cambiamenti eseguiti fanno parte dell’applicazione
- Il build di integrazione può essere eseguito manualmente oppure automaticamente con tool come TeamCity, che, eseguendo un polling continuo sul repository per determinare se ci sono stati aggiornamenti, fa partire il build quando necessario

## Conflitti (1)

- Se ci sono conflitti a causa di due aggiornamenti che si accavallano, solitamente la cosa viene notata dal secondo che esegue commit
- Significa che dal suo ultimo update, qualcun altro ha eseguito un commit
- Il caso viene risolto localmente con un nuovo update (ed eventuali adattamenti) prima del commit

## Conflitti (2)

- Se il conflitto non si manifesta durante il commit (perché non sono stati toccati gli stessi file), ma esiste comunque una inconsistenza, questa viene scoperta durante il build di integrazione
- In entrambi i casi il problema viene scoperto velocemente
- La cosa più urgente da fare in questi casi è riparare il build

## Conflitti (3)

- In un ambiente di integrazione continua non si dovrebbe mai lasciare un “failed build” troppo a lungo
- Un buon team dovrebbe avere più di un build corretto al giorno
- Ognuno può quindi sviluppare a partire dall’ultima versione del build, mantenendo quindi il delta tra sviluppo e successiva integrazione il più piccolo possibile

## Elementi di CI

- Quanto visto nell’esempio precedente dimostra CI (continuous integration) nel lavoro di tutti i giorni
- Vediamo quali sono gli elementi essenziali affinché tutto questo possa avvenire in modo naturale e senza grossi problemi

## Elemento 1: Un solo repository

- I progetto software richiedono la gestione di parecchi file: utilizzare un sistema di versioning control
- Fare in modo che sia accessibile a tutti, anche da remoto
- Nel repository dovrebbe esserci tutto quanto server per eseguire il build, inclusi script, property file, librerie, ecc.
- Regola: dev'essere possibile eseguire un check out su una macchina "verGINE" ed poter subito eseguire un build

## Elemento 2: Build automatico (1)

- Build significa trasformare i sorgenti in un sistema funzionante
- Questo può essere un processo complicato che include compilazione, spostamento di file, caricamento di uno schema di DB, ecc.
- Tutto questo può essere automatizzato ed è buona cosa che lo sia, perché fa risparmiare parecchio tempo

## Elemento 2: Build automatico (2)

- Ambienti di build automatico ne esistono parecchi: Make in Unix, Ant, Maven, Gradle nel mondo Java, Nant o MSBuild per .NET, ecc.
- A dipendenza delle necessità si deve poter creare build in modo condizionale, avendo a disposizione diversi target (build con codice di test integrato, con verifiche diverse, ecc.)
- Il build può essere creato via IDE, ma dev'essere in ogni caso possibile creare un master sul server, senza IDE

## Elemento 3: Build self-testing

- Build non significa solo compilazione e link, un programma compilato correttamente può avere errori di esecuzione
- Il modo migliore per verificare il funzionamento è inserire la chiamata ai test nel processo di build
- Se un test non passa, il build deve fallire

## **Elemento 4: Si integra ogni giorno (1)**

- L'integrazione è anche un mezzo per comunicare agli altri sviluppatori quali modifiche abbiamo apportato al codice
- Integrando regolarmente si scoprono prima eventuali conflitti di sincronizzazione tra sviluppatori e si possono correggere velocemente
- Conflitti che rimangono irrisolti per giorni o settimane, sono difficili da riparare

## **Elemento 4: Si integra ogni giorno (2)**

- Regola: ogni sviluppatore dovrebbe eseguire un commit almeno una volta al giorno
- Commit frequenti (anche più volte al giorno) incoraggiano lo sviluppatore a suddividere il suo lavoro in fasi di alcune ore l'una. Questo permette di “sentire” il progredire del progetto

## Elemento 5: Mantenere il build veloce (1)

- Un elemento essenziale dell'integrazione continua è il feedback veloce
- Le “guidelines” di eXtreme Programming parlano di un massimo di 10 minuti
- Molto spesso se c’è un problema a mantenere il build a 10 minuti, questo è provocato dai test, soprattutto quelli che fanno uso di servizi esterni, come DB

## Elemento 5: Mantenere il build veloce (2)

- In caso di build troppo lunghi, si deve fare in modo di organizzare il processo a tappe (*staged build*, o *build pipeline*)
- Si parte da un “commit build” che deve durare al massimo 10 minuti, poi si possono organizzare passaggi successivi.
- Il commit build viene usato come punto di riferimento per il ciclo di integrazione continua

## Elemento 5: Mantenere il build veloce (3)

- **Esempio: two stage build**

- Il commit build si occupa della compilazione e dell'esecuzione dei test più essenziali, utilizzando oggetti "Mock" per i test sul DB
- Una volta eseguito questo (nei 10 minuti massimi) esiste un build non affidabile al 100%, ma sufficientemente sicuro da poter permettere agli sviluppatori di basare le proprie modifiche su questo ultimo build
- La seconda fase prevede l'utilizzo di tutti i test. A questo punto può anche durare alcune ore
- Se nella seconda fase si riscontrano problemi, si cerca di creare nuovi test per la prima fase che permettano di scoprire in anticipo quanto si è trovato di problematico

## Elemento 6: clonare l'ambiente di produzione (1)

- È importante poter eseguire tutti i test in un ambiente il più possibile simile a quello di produzione
- Ogni differenza può essere motivo di errore in produzione
- Clonare significa avere le stesse versioni del software, del sistema operativo, del DB, stesse librerie, stesso HW
- Ci sono dei limiti (HW troppo caro), ma spesso i vantaggi li superano ampiamente

## **Elemento 6: clonare l'ambiente di produzione (2)**

- Quando l'applicazione deve poter andare in produzione su ambienti diversi (esempio: applicazione desktop) è quasi impossibile provare tutte le combinazioni
- In questi casi possono però aiutare gli ambienti virtuali (VMWare, Parallels, ecc.)

## **Elemento 7: Rendere accessibile il build (1)**

- Uno dei maggiori problemi nello sviluppo del software è sapere se si stanno sviluppando le funzionalità realmente desiderate dal committente
- Le persone trovano più semplice vedere qualcosa di incompleto, ma che permetta di capire meglio cosa si desidera e come esprimere
- I processi agili si basano su questo e traggono vantaggio da questo tipico comportamento umano

## **Elemento 7: Rendere accessibile il build (2)**

- In queste situazioni è importante che ogni sviluppatore abbia facile accesso all'ultimo build, o al build della sua ultima iterazione
- Lo scopo è quello di poterlo usare per dimostrazioni, test, o anche unicamente per verificare cosa è cambiato negli ultimi giorni
- Il repository deve essere organizzato in modo che ci sia una chiara sequenza storica dei build

## **Elemento 8: Visione dello stato (1)**

- CI serve anche alla comunicazione, perciò è importante che ognuno possa verificare lo stato del sistema e i cambiamenti effettuati
- I programmi di CI hanno un'interfaccia Web che permette di accedere alla storia dei cambiamenti, sapere chi ha fatto le modifiche, ecc.
- Il vantaggio di un'interfaccia Web consiste nel fatto che anche sviluppatori localizzati altrove hanno facile accesso alle informazioni

## Elemento 8: Visione dello stato (2)

- Interfaccia Web di TeamCity

Collapse All | Expand All 0 build(s) running.

**CobraWunelli** hide project

- = **CobraWunelli Maintenance Build Configuration**  
#build.563-2 Tests passed: 159 | No artifacts Changes (1) 10 Feb 17:56 (7m:37s)  
Idle Run
- = **CobraWunelli Trunk Build Configuration**  
#build.680 Tests passed: 191 | No artifacts Changes (1) 25 Feb 15:33 (12m:09s)  
Idle Pending (1) Run

Sandro Pedrazzini Integrazione continua 37

## Elemento 9: Deployment automatico

- Un elemento di integrazione continua è il deployment automatico
- Questo può essere utile soprattutto se si hanno diversi deployment in ambienti diversi: automatizzare significa evitare facili fonti di errore
- Se è compresa la funzionalità di deployment in produzione, una capacità interessante è quella del rollback automatico alla versione precedente: questo permette di eliminare la “tensione” tipica del deployment

## Benefici della CI (1)

- Minor rischio
  - Ricordo di progetti senza integrazione continua: progetto terminato, ma incognita dell'integrazione
  - Difficile prevedere il tempo necessario di un'integrazione prevista solo alla fine del progetto
  - CI permette di sapere in ogni momento a che punto si è con il progetto e con gli errori

## Benefici della CI (2)

- Errori
  - CI non ci permette di eliminare gli errori, ma ci rende più semplice il compito di trovarli
  - Quando si introduce un bug nel sistema, se lo si scopre in fretta, diventa semplice anche toglierlo
  - Inoltre l'errore può essere solo introdotto in poche ore di lavoro dall'ultimo build stabile
  - Si eliminano i bug accumulati nel tempo

## Benefici della CI (3)

- Deployment più frequente
  - Grossa barriera che CI permette di eliminare
  - Permette di mostrare più rapidamente nuove features all'utente finale e ottenere di conseguenza un feedback più veloce
  - Attraverso feedback più veloce, utente e sviluppatore migliorano il loro rapporto di collaborazione
  - Si accorcia la distanza che esiste tipicamente tra sviluppatore e utente, decisiva per uno sviluppo software di successo

## Integrazione con le altre pratiche (1)

- L'utilizzo di CI si integra bene con altre pratiche di progettazione e sviluppo trattate
  - Test di unità
  - Utilizzo di standard nel codice
  - Refactoring
  - Cicli di sviluppo corti
  - Appartenenza comune del codice

## Integrazione con le altre pratiche (2)

- **Test di unità**

- Chi sviluppa, dovrebbe aggiungere codice di test al proprio codice (unit testing)
- Il test dovrebbe essere richiamato dopo ogni cambiamento
- Con CI il test viene richiamato automaticamente ad ogni build, quindi ad ogni modifica nel repository (regression test)

## Integrazione con le altre pratiche (3)

- **Utilizzo di standard**

- In ogni progetto si definiscono delle *guidelines* da seguire, in modo che l'intero codice segua gli stessi "standard"
- Spesso il controllo dell'aderenza allo standard è un processo manuale
- Con CI si può inserire una serie di analisi statiche del codice nello script di build, in grado di generare un report

## Integrazione con le altre pratiche (4)

- **Refactoring**

- Refactoring significa adattare il codice e la sua struttura interna senza modificare la funzionalità
- Uno degli scopi è quello di facilitare la manutenzione del codice
- CI assiste lo sviluppatore permettendo la chiamata di tool di inspection del codice all'interno del build, eseguito ad ogni modifica del repository

## Integrazione con le altre pratiche (5)

- **Cicli brevi**

- Significa che gli utenti devono avere a disposizione l'ultima versione del software funzionante il più spesso possibile
- CI si adatta bene a questa pratica, perché si integra più volte al giorno e ogni integrazione genera virtualmente un nuovo release
- Quando un sistema di integrazione continua è installato, un nuovo release viene generato con il minimo degli sforzi

## Integrazione con le altre pratiche (6)

- **Appartenenza comune (*collective ownership*)**
  - Ogni sviluppatore può lavorare a qualsiasi parte del sistema
  - Questo impedisce che ci sia un solo sviluppatore con conoscenze specifiche di un determinato argomento
  - CI aiuta questa pratica assicurando aderenza agli standard e richiamando continuamente i test di regressione

## Ridurre i rischi (1)

- **CI aiuta nel mantenere i rischi sotto controllo, permettendo di scoprire i problemi appena questi si manifestano**
- **Con CI e le pratiche correlate si riesce a creare una rete di qualità, che ci permette di fornire software di qualità più velocemente**

## Ridurre i rischi (2)

- Consideriamo i seguenti rischi
  - Software non deployable
  - Difetti scoperti tardi
  - Mancanza di visibilità del progetto
  - Software di bassa qualità
- Non si tratta di rendere attenti verso questi rischi (sono tutti noti), ma di permetterne la gestione

## Ridurre i rischi (3)

- Software non “deployabile”, impossibile creare il build

- Riesco a creare il build solo sulla mia macchina

In questo caso è importante sottolineare che il build dev'essere creato in modo indipendente dalla macchina, IDE utilizzato, configurazione specifica, ecc.  
È importante avere un server di integrazione, che usi uno script di build (ant) indipendente.

- Sincronizzazione con il DB

I test devono poter girare utilizzando l'ultima versione dello schema di DB. Lo schema di DB e i suoi dati minimi indispensabili devono trovarsi nel repository.  
I test devono essere in grado di eseguire un “drop” del DB e poi ricrearlo nuovo.

## Ridurre i rischi (4)

- **Difetti scoperti tardi**

- **Regression test**

Sappiamo che dobbiamo avere suite di test nel nostro progetto.  
Basta aggiungere la chiamata di questi test nello script di build, in questo modo verranno eseguiti dal server di integrazione ad ogni modifica.

- **Test coverage**

Esistono tool per verificare la percentuale di copertura dei test.  
Questi tool possono essere associati al processo di CI.

## Ridurre i rischi (5)

- **Mancanza di visibilità**

- **Informazioni**

È necessario che ogni sviluppatore nel progetto sia informato su ogni singola modifica effettuata al progetto.  
Al processo di CI può essere associata una notifica via email sulle modifiche effettuate o sugli eventuali problemi.

- **Visualizzazione grafica della struttura**

Se si desidera avere a disposizione l'ultima versione grafica della struttura del software (UML class diagram), lo si può fare inserendo nel sistema di CI la generazione a partire dall'ultima versione (esempio: Doxygen).

## Ridurre i rischi (6)

- **Software di bassa qualità (1)**

- **Aderenza del codice a standard predefiniti**

L'aderenza agli standard viene spesso controllata manualmente. In realtà esistono tool come Checkstyle, PMD o anche Sonar, che permettono di fare delle verifiche statiche del codice a partire da regole. Questi possono essere integrati al sistema CI.

- **Aderenza all'architettura**

Anche a livello di architettura ci possono essere delle guideline da seguire, ad esempio, il codice del data layer che non accede al codice del business layer, ecc.

Anche queste possono essere controllate da tool (JDepend, NDepend, ecc.) integrabili nel processo di CI.

## Ridurre i rischi (7)

- **Software di bassa qualità (2)**

- **Duplicazione del codice**

Codice duplicato rende più difficili le modifiche e la manutenzione del progetto.

È difficile scoprire dove abbiamo duplicazione di codice all'interno del progetto. Più il progetto è grande, più sviluppatori lavorano e maggiore è la probabilità di avere codice in più parti che esegue la stessa funzionalità.

Anche in questo caso esistono tool (utility CPD del tool di analisi metriche PMD, Simian, ecc.) che possono essere integrati nel processo di CI

## Come introdurre CI (1)

- Tutte le pratiche viste fin qui servono a trarre il massimo beneficio da CI
- Per iniziare non serve però applicarle tutte assieme
- Il primo passo è senza dubbio quello dell'automazione del processo di build
  - Mettiamo il progetto sotto il controllo di un sistema di gestione dei sorgenti (esempio: Subversion)
  - Facciamo in modo che con un unico comando si possa creare un build

## Come introdurre CI (2)

- Il passo successivo potrebbe essere quello di introdurre suite di test nel build automatico
- Se si hanno già test di unità nel sistema, la cosa è molto semplice, basta richiamarli durante il build
- Inizialmente il build verrà fatto partire a mano. Poi, si potrà richiamarlo automaticamente, con uno script, una volta al giorno per il nightly build
- Come ultimo passaggio, possiamo installare un server di integrazione con un software di CI (esempio: TeamCity)