Appunti: “Practices Of An Agile Developer”

Manuel Paccagnella
Testo di Venkat Subramaniam & Andy Hunt
http://www.pragprog.com/titles/pad/practices-of-an-agile-developer

13 novembre 2009

Indice

Capitolo 1

Agile Software Development

> No matter how far down the wrong road you've gone, turn back.
> —Turkish proverb

Per sua natura ogni progetto software è una storia a sè e continuamente mutevole. E' un bersaglio in movimento, e per tale motivo una caratteristica molto importante sia del singolo che del team è l'*agilità*: saper **adattarsi** rapidamente alla situazione.

Lo spirito dell'agilità è andare diretti all'essenziale (secondo il principio di Pareto) e focalizzarsi sui risultati, perchè alla fine i risultati sono tutto quello che conta. E questo essenziale è: *persone, collaborazione, responsività e software funzionante.*

> **The Agile Manifesto** We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools
> - Working software over comprehensive documentation
> - Customer collaboration over contract negotiation
> - Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.
>
>
> See http://agilemanifesto.org for more information.

Ma occorre anche un processo che permetta di applicare il ciclo di Deming, quindi **ci si focalizza su obiettivi ben definiti e *misurabili*, combinandolo con dei *cicli di feedback* frequenti per apportare miglioramenti al processo di sviluppo**. Questi cicli sono *iterazioni* e normalmente lunghi circa una settimana nella quale si identificano e si realizzano determinate features.

> AGILE DEFINITION: **Agile development uses feedback to make constant adjustments in a highly collaborative environment.**

Lo stile di sviluppo agile, presuppone che i componenti del team abbiano un atteggiamento *professionale* (a ognuno sta a cuore fare bene il proprio lavoro). Se non è questo il caso, è meglio un approccio più plan-based, lento e complesso (maggior controllo).

L'essenza dell'*agile software development* è la consapevolezza che ogni progetto è un'attività continuativa che richiede che le attività del processo di sviluppo siano distribuite lungo tutto il ciclo di vita del software (raccolta dei requisiti, testing, integrazione, deployment, analisi del feedback, training, ecc).

> Continuous development, not episodic.

Il processo di sviluppo e manutenzione del software è senza dubbio un *sistema complesso*, sempre sull'orlo dell'entropia, e come tale ha bisogno di cure continue su base *as-needed*.

Quello che NON è l'agile è crisis-management, perchè a differenza di quest'ultimo incorpora una forte e continua componente di awareness che permette di riparare i problemi non appena sorgono, non quando ormai sono diventati talmente grossi da costituire una crisi che deve essere gestita subito e a dispetto di tutto il resto.

Capitolo 2

Beginning Agility

La metodologia di tipo agile è differente dalle precedenti. Invece di porre il focus su ruoli, responsabilità e processi, lo pone sulle persone. Il focus è sull'*essenziale*: gli sviluppatori come persone (con proprie abilità, competenze e personalità) che lavorano a contatto con i clienti per soddisfarne al meglio le esigenze.

> Instead of relying on Gantt charts and stone tablets, agility relies on people.

Non si tratta di affidarsi totalmente ai capricci delle persone, ma di approfittare dei punti di forza di ogni componente del team senza spersonalizzare ogni lavoro (che è deleterio per la motivazione e lo sviluppo).

Il focus sulle persone e sull'essenziale significa che lo sviluppo del software è più un **atteggiamento** che un qualche particolare metodologia o strumento.

> Software development doesn't happen in a chart, an IDE, or a design tool; it happens in your head. But it's not alone.

2.1 1. Work for outcome (FOCUS)

Il principale e più importante obiettivo da tenere sempre in mente è quello di creare un buon prodotto. Non serve incolpare la gente e perseguirla. Il *focus* deve essere sempre sull'obiettivo: un buon prodotto e la risoluzione tempestiva dei problemi non appena si presentano.

> If you go to an agile team member with a complaint, you'll hear, "OK, what can I do to help you with this?" Instead of brooding over the problem, they'll direct their efforts toward solving it. Their motive is clear; it's the outcome that's important, not the credit, the blame, or the ongoing intellectual superiority contest.

Fare questo in prima persona, chiedere che cosa si può fare per aiutare, indica la volontà di far parte della soluzione, non del problema.

> Blame doesn't fix bugs. Instead of pointing fingers, point to possible solutions. It's the positive outcome that counts.

Questo focalizzarsi sul risultato più che sul processo è in contrasto con l'approccio classico alla *software engineering* nel quale la misura più importante è l'aderenza del team ad un processo di sviluppo provato. Sfortunatamente questo non garantisce che il risultato sia buono perchè il focus diventa il processo e gli strumenti invece che l'obiettivo di tali attività.

2.1.1 What it feels like

Un grosso errore è una grande opportunità di imparare. Nessuna caccia alle streghe. Si ha la sensazione di lavorare in un team in cui tutti sono alleati e si supportano l'un l'altro.

2.2 2. Quick fixes become Quicksand (COMPRENSIONE)

> Don't fall for the quick hack. Invest the energy to keep code clean and out in the open.

Le "toppe" possono essere seducenti sotto pressione, ma non bisogna dimenticare il passo essenziale: **studiare la radice del problema**. Occorre capire qual'è la causa e lavorare su quella, studiare il codice e il suo funzionamento. Diversamente, toppa dopo toppa, ad un certo punto il sistema collassa e qualsiasi operazione di enhancement o bug-fixing diventa un incubo.

> Fix the problem, not the symptom.
>
> Don't live with broken windows!

E' necessario maturare una vera *comprensione del sistema*, del suo funzionamento e del suo dominio applicativo.

> Shallow hacks are the problem—those quick changes that you make under pressure without a deep understanding of the true problem and any possible consequences.

Avere parti di codice "opache" o che nessuno comprende è da evitare accuratamente perchè chiaramente inibisce l'agilità. Alcune pratiche per coltivare la comprensione:

- **Collective Code Ownership** (+ *Code Reviews*)
- **TDD**

2.2.1 What it feels like

Si ha la sensazione che il codice sia ben fatto, che non ci siano angoli ciechi e nessuna porzione di codice recintata con il filo spinato.

Anche se non è necessario che ognuno conosca e comprenda ogni singola linea di codice, *una buona conoscenza della struttura e del funzionamento dell'applicazione è essenziale.* Quindi il progetto deve essere opportunamente **modulare** e ogni componente del team deve avere una comprensione ad alto livello del funzionamento di ogni modulo e di come questo interagisce con il resto dell'applicazione. Poi la conoscenza approfondita si ha ovviamente nel modulo a cui si lavora.

2.3 3. Criticize ideas, not people (COLLABORAZIONE)

> Criticize ideas, not people. Take pride in arriving at a solution rather than proving whose idea is better.

Quando si parla di soluzioni, è facile diventare emotivi riguardo ad esse. Quando sembrano esserci dei difetti, le scelte sono essenzialmente tre:

1. **Criticare la persona.** Non aiuta il suo apprendimento, va contro lo spirito di collaborazione e probabilmente inibirà qualsiasi suo successivo contributo.

2. **Criticare la soluzione.** Un pò meglio, ma può implicare il primo punto. E poi può essere una lama a doppio taglio: l'accusatore potrebbe essere lui stesso in errore e ricevere critiche a sua volta.

3. **Chiedere chiarimenti.** In modo che l'interessato arrivi da solo a scoprire il problema (elimina i difetti della seconda soluzione), incoraggia l'apprendimento, la collaborazione e la comprensione. Stimola una *conversazione* non una *discussione*.

> Keep It Professional, Not Personal.

E' fondamentale capire l'atteggiamento di fondo, perchè è questo il sale e l'ingrediente fondamentale della *collaborazione produttiva*. Occorre affrontare i problemi con calma, professionalità e senza accusare nessuno. Il focus deve essere sul prodotto e sulla collaborazione, sul **problem-solving**. Deve potersi creare un'atmosfera rilassata, quasi familiare in cui ognuno è pronto a fare ciò che può per aiutare gli altri a raggiungere l'obiettivo e migliorare il team stesso (sia per quanto riguarda i processi che il morale). Ognuno dovrebbe avere a cuore lo sviluppo sia proprio che del resto del team.

> Negativity kills innovation.

L'ambiente e l'atmosfera deve essere tale da *incoraggiare* i suggerimenti e i contributi. Se diversamente si corre il rischio di essere ridicolizzati o di perdere la faccia, nessuno suggerirà più nulla. La morte dell'innovazione e della collaborazione.

> We all are capable of generating excellent, innovative ideas, and we are all equally capable of proposing some real turkeys.
>
> "You don't have to be great to get started, but you have to get started to be great."—Les Brown

2.3.1 What it feels like

Atmosfera di collaborazione, di professionalità e di supporto verso l'obiettivo di avere un prodotto funzionante e che rispetta le esigenze del cliente in tempo utile.

Possibilità di proporre soluzioni senza venire criticati, e discussione di pregi e difetti in maniera obiettiva e che sia utile a tutti quanti.

2.4 4. Damn the torpedoes, go ahead (CORAGGIO)

Ogni buon piano che comporta sostanziali vantaggi presenta anche rischi. Per attuare un piano del genere occorre *coraggio* e *fare la cosa giusta*. Adottare la soluzione più semplice e sicura non sempre comporta vantaggi (a volte invece porta svantaggi).

Se ci si accorge di aver commesso un errore, o che a fare un errore è stato qualcun altro, occorre farsi avanti e chiedere aiuto o proporre una soluzione, anche se questo non sempre è facile e comporta un ritardo nei tempi di consegna.

Per esporre il problema, è fondamentale presentare delle *valide ragioni*. Qui le capacità di comunicazione sono essenziali.

L'atteggiamento propositivo, orientato all'obiettivo e la volontà di fare la cosa giusta, se portati avanti con **coraggio e onestà** suscitano rispetto e contribuiscono a migliorare il progetto e il team intero.

> Do what's right. Be honest, and have the courage to communicate the truth. It may be difficult at times; that's why it takes courage.

2.4.1 What it feels like

Mostrare coraggio non è facile, ma è necessario per rimuovere degli ostacoli che con l'andare del tempo possono peggiorare e condannare l'intero progetto.

Capitolo 3

Feeding Agility

Essere nel campo dell'IT comporta **apprendimento continuo**. Occorre stare al passo con le nuove innovazioni tecnologiche e metodologiche se si vuole rimanere competitivi. E questa è essenzialmente una responsabilità personale, anche se il datore di lavoro dovrebbe supportare attivamente la cosa.

3.1 5. Keep up with change (APPRENDIMENTO)

> "Non c'è nulla di permanente, ad eccezione del cambiamento."— Eraclito

La tecnologia avanza a ritmo serrato, ed è essenziale rimanere al passo, essere curiosi e *apprendere* in modo continuativo. Così facendo, visto che gli sviluppi per lo più sono incrementali, continuare a imparare non è poi così difficile. Se invece ci si ferma, e si cerca di rimettersi al passo dopo anni... buona fortuna!

Che strumenti si possono utilizzare per tenersi al passo?

- **Imparare iterativamente e in modo incrementale**: annotare termini non familiari o tecnologie di interesse e schedulare del tempo ogni giorno per investigare, apprendere e sperimentare.
- **Rimanere informati sulle novità**: scegliere alcuni dei blog migliori e tenersi aggiornati.
- **Frequentare conferenze**: per imparare direttamente dagli esperti.
- **Leggere voracemente**: libri tecnici, riviste, libri non tecnici, news.

> Keep up with changing technology. You don't have to become an expert at everything, but stay aware of where the industry is headed, and plan your career and projects accordingly.

Non occorre diventare esperti in ogni cosa, nè investire troppo tempo in tecnologie che potrebbero non sfondare. Occorre usare il buon senso, essere equilibrati e avere gli occhi aperti.

3.1.1 What it feels like

Si è aggiornati sulle nuove tecnologie, si sa dove l'industria si sta muovendo e se si deve cambiare tipologia di lavoro, si è in grado di farlo.

3.2 6. Invest in your team (SCAMBIO DI CONOSCENZE)

Non basta che qualcuno sia molto intelligente e preparato, i migliori risultati si ottengono quando l'intero team apprende e migliora con il tempo. Il team, nel quale le persone hanno capacità e livelli di competenza differenti, è l'ambiente ideale per stimolare e coltivare l'apprendimento.

3.2.1 Brown-bag sessions

Un'idea è quella delle *brown-bag session*: un'oretta alla settimana (di solito all'ora di pranzo, in un qualunque giorno escluso lunedì e venerdì) in cui un "relatore" porta qualcosa di interessante al team (demo di qualche tool, discussione su nuovi temi, esercitazioni, book-club, ecc).

Il relatore parla per una quindicina di minuti e poi la discussione si apre per scambiare conoscenze e pensieri, portando esempi (anche dal proprio progetto). E' utile pianificare anche delle continuazioni se necessario.

> Raise the bar for you and your team. Use brown-bag sessions to increase everyone's knowledge and skills and help bring people together. Get the team excited about technologies or techniques that will benefit your project.

L'ingrediente fondamentale dev'essere l'intimità, ovvero un'atmosfera informale che stimola la discussione. E' anche importante aderire ad una schedule in modo costante:

> Stick to a regular schedule. Constant, small exposure is agile. Infrequent, long, and drawn-out sessions are not.

Gli argomenti discussi dovrebbero restare sul generale. Rilevanti per il progetto ma la brown-bag session non deve trasformarsi in una sessione di design altrimenti si trasforma in un meeting di lavoro:

> Brown-bag sessions aren't design meetings. Overall, you want to focus on discussing general topics that are relevant to your application. Solving specific issues is usually better left to a design meeting.

3.2.2 What it feels like

L'intero team diventa più "intelligente", è al corrente delle novità nell'industria e diventa in grado di discutere potenziali applicazioni nei progetti e difetti di nuove tecnologie.

3.3 7. Know when to unlearn (DIS-APPRENDIMENTO)

In un mondo in costante cambiamento, un valore fondamentale dell'approccio agile è l'*adattamento*. Questo significa non solo imparare nuove tecnologie, ma anche *disimpararne* di vecchie che possono limitare la propria efficacia ed efficienza nel lavorare con le nuove.

Perciò è necessario approcciare ogni nuova tecnologia con la *beginner's mind*: senza pregiudizi e cercando di limitare il peso che le vecchie (e non più applicabili) conoscenze proiettano sulle nuove. L'esplorare nuovi strumenti o tecniche con lo stesso modo di vedere maturato con quelle vecchie può inibirne il corretto apprendimento e applicazione. Ad esempio: per imparare la programmazione orientata agli oggetti si deve cambiare punto di vista e "dimenticare" il paradigma procedurale, altrimenti si continurà a scrivere codice procedurale.

In altre parole, nell'imparare nuove tecnologie immergervisi senza portarsi dietro il bagaglio delle vecchie (usare subito e solo nuovi tool e librerie senza riutilizzare i vecchi).

> Learn the new; unlearn the old. When learning a new technology, unlearn any old habits that might hold you back. After all, there's much more to a car than just a horseless carriage.

3.3.1 What it feels like

Le nuove tecnologie possono spaventare perchè richiedono di imparare cose nuove, ma è giusto che sia così e le vecchie conoscenze che si possiedono vanno usate come *base*, non come stampella...

3.4 8. Question until you understand (SCAVARE FINO ALLA RADICE DEI PROBLEMI)

Occorre arrivare alla *radice* di un problema, e per fare questo spesso non è sufficiente accontentarsi delle risposte che normalmente si ricevono: bisogna continuare a indagare chiedendo, "**perchè?**"

Diversamente si ignora un problema potenzialmente grave o ci si mette una toppa per poi soccombere schiacciati dal suo peso in un futuro più o meno prossimo.

Continuare a chiedere (e chiedersi) "*perchè?*" fino ad arrivare alla vera causa.

> Keep asking Why. Don't just accept what you're told at face value. Keep questioning until you understand the root of the issue.

3.4.1 What it feels like

E' come scavare alla ricerca di una gemma preziosa. Si cerca e indaga tra materiali scorrelati, sempre più in profondità fino a scoprirne la vera causa. Si è consapevoli di aver scoperto la radice del problema e non solo un suo sintomo.

Scavare in profondità!

3.5 9. Feel the rhythm (TIME-BOXING)

Non è una buona idea NON avere alcun piano e lasciare ogni cosa al caso senza alcun punto di riferimento. Così come è inutile pianificare ogni cosa nei minimi dettagli.

I progetti agili hanno un *ritmo*: usano il tempo per stabilire dei cicli con i quali sincronizzarsi. C'è il detto secondo il quale il tempo esiste per non far accadere le cose tutte insieme, questo vale sicuramente per i progetti agili: *il tempo esiste per non far succedere tutto insieme o a caso.*

Un paio di ritmi fondamentali sono:

- **Il giorno**: idealmente dovrebbe essere l'unità di lavoro fondamentale, alla fine della quale non dovrebbe esserci nulla in sospeso (codice committato e build automatico con successo). Diversamente, va considerata serialmente la possibilità di *buttare via il codice in sospeso e ricominciare d'accapo.* Anche se si butta via il codice, ci sarà l'esperienza e il ragionamento maturato che serviranno ad arrivare ad una soluzione migliore e più velocemente. Ne consegue che il problema da affrontare ogni giorno deve essere della giusta dimensione e difficoltà. [*Time-boxing con una hard-deadline.*]

- **L'iterazione**: (tipicamente da 1 a 4 settimane, 2 in media) è importante avere iterazioni di lunghezza consistente.

  > Tackle tasks before they bunch up. It's easier to tackle common recurring tasks when you maintain steady, repeatable intervals between events.

3.5.1 Time-boxing

Un buon *time-box* ha queste caratteristiche:

- non può slittare (hard time-boxing)
- ha una data *precisa* associata
- ha un obiettivo *ben definito*

Usando l'approccio del time-boxing le date rimangono fisse, ma la quantità di lavoro terminata all'interno di un time-box è flessibile. Quando arriva la deadline l'iterazione è finita, e sono le *features* che possono slittare.

> A hard deadline forces you to make the hard choices. You can't waste time on philosophical discussions or features that are perpetually 80% done. A time box keeps you moving.

Inoltre, stabilire obiettivi piccoli e ben definiti in ogni time-box in modo che vengano raggiunti mantiene alta la motivazione e fornisce un senso di *successo* che va celebrato adeguatamente (pizzata, cena fuori, gita a Cortina, ecc).

3.5.2 What it feels like

Si lavora con un ritmo consistente e definito: ci sono dei cicli di lavoro stabili e prevedibili con i quali ci si sincronizza e che scandiscono il ritmo di lavoro aiutando nel prendere decisioni su cosa, quanto e come lavorare.

> It's easier to dance when you know when the next beat falls.

Capitolo 4

Delivering What Users Want

> "No plan survives contact with the enemy."—Helmuth von Moltke

E il nemico è il *cambiamento*. **Essere veramente agili significa saper riconoscere e adattarsi al cambiamento** (tra le altre cose) in ogni fase e attività del ciclo di vita del software (raccolta dei requisiti, design, ecc). Solo così si sarà in grado di consegnare un prodotto soddisfacente rispettando i vincoli di tempo e budget.

4.1 10. Let customers make decisions (COINVOLGIMENTO CLIENTE)

I clienti devono essere coinvolti il più possibile nel processo decisionale riguardo la realizzazione del progetto e devono avere già molto presto una qualche demo sulla quale possa fornire *feedback*.

La scelta è tra coinvolgere presto il cliente ad un costo risibile, o coinvolgerlo solo più tardi risparmiandosi la fatica ma a un costo (quasi sicuro) molto più alto quando si scopre di dover fare pesanti modifiche per soddisfare i requisiti (quelli reali e corretti).

> Decide what you shouldn't decide.

Le decisioni di business le fanno chi gestisce il business.

> If the issue at hand affects the behavior of the system, or how it'll be used, take it to the business owner.

Quando si parla con il cliente, bisogna:

- presentare tutte le opzioni, con i relativi pro e contro
- rendere espliciti i compromessi di una soluzione rispetto alle altre

- l'impatto di ogni decisione sui tempi di consegna e sul budget

> Let your customers decide. Developers, managers, or business analysts shouldn't make business-critical decisions. Present details to business owners in a language they can understand, and let them make the decision.

Nel caso la decisione da parte del cliente sia un "*non lo so*", prendere una decisione ma preparare il codice ad essere eventualmente cambiato e sottoporgli il prima possibile una demo sulla quale valutare.

Le decisioni prese (e il ragionamento che ha condotto ad esse) devono essere poi **registrate** (documenti, wiki, serie di email, ecc). Porre particolare attenzione che qualunque metodo per registrare decisioni non sia troppo oneroso da mantenere (perchè *deve* essere fatto).

4.1.1 What it feels like

Applicazioni business-critical devono essere realizzare in *partnership* con il business-owner. Deve essere una onesta relazione lavorativa.

4.2 11. Let design guide, not dictate (DESIGN)

Il design è una fase essenziale del processo di sviluppo: aiuta a investigare i dettagli del sistema, le interrelazioni tra i sotto-sistemi, a capire meglio il dominio applicativo e dirige verso una buona implementazione.

Un certo livello di progettazione è necessario per capire e valutare qual'è il modo migliore di procedere, ma il BDUF (Big Design Up-Front) è una pratica inutile e una perdita di tempo. Il software è più simile ad un giardino che ad un ponte o un palazzo: miriadi di sorprese, cambiamenti di direzione ed errori di valutazione attendono dietro l'angolo e investire eccessivamente in un piano iper-dettagliato che sarà quasi sicuramente da cambiare pesantemente è inutile.

> "There's no sense being exact about something if you don't even know what you're talking about." —John von Neumann

L'approccio agile suggerisce invece di **cominciare presto la codifica, non appena si è fatto quel tanto di design che basta per individuare la direzione migliore verso cui andare**. Persino nell'analogia delle costruzioni, una trave viene tagliata un pò più lunga del necessario per poi essere accorciata in base alle esigenze.

Una componente fondamentale dell'approccio agile è infatti l'abbracciare il cambiamento e lo sfruttare con costanza ogni *feedback* per dirigere i

propri sforzi in maniera ottimale. Nel caso del design, la prima progettazione riflette la comprensione *attuale* del problema, che evolverà nel tempo con l'acquisizione di nuova esperienza.

Il design dovrebbe consistere in alcuni diagrammi chiave che individuino l'architettura, l'organizzazione delle classi e le interazioni fondamentali, in modo da poter discutere le varie opzioni e compromessi necessari. Una volta individuato il modo migliore per procedere, partire con la codifica e rimanere agili: ovvero essere pronti al cambiamento (per questo è necessario mantenere sempre la base di codice in salute con una serie di pratiche: TDD, refactoring, CI, code reviews, ecc).

Si distinguono due tipi fondamentali di design:

- **Strategico**: è il design up-front (architetturale) che deve produrre più che altro una mappa che indica in che direzione andare. Non ha senso specificare i particolari quando ancora non si conosce in che territori ci si troverà a viaggiare. E' un punto di partenza che verrà modificato e rifinito con l'andare del tempo, l'esplorazione del dominio e il maturare dell'esperienza. In questa fase può essere utile specificare le classi non in termini di "tattica" quanto di *responsabilità* [**CRC cards**].
- **Tattico**: la progettazione in particolare, in una situazione e area del sistema specifica. Tipicamente, si parte dal *comportamento* richiesto per arrivare alla struttura [**TDD, BDD**]

Come giudicare se un design è buono o cattivo? Realizzare il codice! Se apportarvi piccole modifiche richiede grandi cambiamenti non è un buon design, diversamente si.

> A good design is a map; let it evolve. Design points you in the right direction. It's not the territory itself; it shouldn't dictate the specific route. Don't let the design (or the designer) hold you hostage.

La progettazione quindi è utile nella misura in cui illumina riguardo l'approccio da adottare nello sviluppo:

> White boards, sketches and Post-it notes are excellent design tools. Complicated modeling tools have a tendency to be more distracting than illuminating.

4.2.1 What it feels like

Un buon design è accurato ma non preciso. Non deve contenere dettagli che possono cambiare. **E' un'intento non una ricetta!**

In altre parole, il design è uno strumento, non la parola finale. La parola finale ce l'ha il codice (la sua espressione nel mondo "reale") ed è necessario

il feedback che ne deriva per verificare la bontà della mappa. Quindi a maggior ragione, pianificare quanto basta per individuare la direzione migliore e partire, stabilendo la tattica migliore di volta in volta.

4.3 12. Justify technology use (SCELTA DELLE TECNOLOGIE)

Le tecnologie da utilizzare non vanno scelte solo in base a quanto "bello" sarebbe utilizzarle o a quanto "bene" figurino nel proprio bagaglio di conoscenze. La decisione deve essere pragmatica e fondarsi sul giusto obiettivo: *questa tecnologia risolve effettivamente un problema? E' utile realizzare meglio il progetto (con meno tempo/costo/fatica)?*

Le domande da porsi sono:

- **Risolve veramente il problema?** Occorre essere sicuri dei vantaggi e di potenziali difetti. Realizzare un prototipo se serve. Basarsi su esperienza (di prima o successiva mano), non sul marketing.
- **E' una tecnologia che comporta un lock-in?** E' importante la *reversibilità* di una scelta tecnologica. Tenere da conto quanto aperta o proprietaria è.
- **Quali sono i costi di manutenzione?**

Nella scelta delle tecnologie non basta considerare le sole features, va considerato l'intero *contesto*.

> Always consider the context.—"Pragmatic Thinking & Learning", Andy Hunt

Inoltre, la scelta va fatta basandosi su un reale bisogno e sull'*esperienza* maturata (necessaria perchè la scelta ponderata).

> **Choose technology based on need.** Determine your needs first, and then evaluate the use of technologies for those specific problems. Ask critical questions about the use of any technology, and answer them genuinely.

4.3.1 Framework

Lo sviluppo di un framework in-house dovrebbe essere sempre l'ultima scelta:

> Don't build what you can download.

Meno codice si scrive, meno codice si deve mantenere!

4.3.2 What it feels like

> A new technology should feel like a new tool that does a better job; it shouldn't become your job.

4.4 13. Keep it releasable (CODICE SEMPRE RILASCIABILE)

E' importante che il codice sia sempre in uno stato consistente e pronto per essere rilasciato (o in forma di demo o di applicazione più o meni utilizzabile).

Nel momento in cui non si rispetta più la regola di lasciare il codice in uno stato consistente si apre la porta a tutta una serie di problemi: modifiche incompatibili, bug che rimangono presenti per lungo tempo, ecc. Questi problemi si accumulano e corrompono il prodotto come una malattia degenerativa rendendo sempre più difficile e costoso riportare il tutto a funzionare correttamente.

Molto meglio avere una *policy* che prescrive di avere il software funzionante e rilasciabile ad ogni push nel repository di riferimento (in questo aiuta prevedere nell'infrastruttura un server di *continuous-integration*).

Ricollegandosi anche al discorso del "ritmo" di sviluppo: al termine di ogni giornata non ci dovrebbero essere feature in sospeso e il software dovrebbe essere funzionante.

Risulta utile adottare un workflow di questo tipo una volta completato un task:

- **Far girare tutti i test localmente**: assicurarsi che il software funzioni come dovrebbe facendo girare localmente *tutti* i test di unità.
- **Recuperare l'ultima versione dei sorgenti**: eventualmente risolvere i conflitti e far girare nuovamente tutti i test per verificare che ogni cosa continui a funzionare correttamente. Questa fase è necessaria per verificare ed integrare eventuali modifiche realizzate da persone diverse che entrano in conflitto tra loro.
- **Push**: una volta verificato il funzionamento e integrato con eventuali altre modifiche si può immettere il tutto nel repository di riferimento.

Può succedere che durante questo processo si scoprano delle modifiche effettuate da altre persone che hanno lasciato il codice in uno stato inconsistente (quando l'errore è il proprio, sistemare personalmente). In tal caso avvertire l'interessato ed eventualmente tutto il team: il sistemare il codice per riportarlo ad uno stato rilasciabile diventa un task ad *alta priorità* (per sistemare, si può considerare la pratica del *pair-programming*). Avere un

sistema di continuous-integration che lo faccia automaticamente (ad eccezione del sistemare il codice) è un'ottima cosa perchè aiuta a far rispettare questo criterio avvertendo tempestivamente tutto il team.

Quando si devono introdurre delle modifiche che alterano significativamente il comportamento del sistema, le possibilità di creare conflitti o lasciare il codice in uno stato inconsistente per diverso tempo sono molto alte. In questo caso il criterio di lasciare sempre il software in uno stato rilasciabile suggerisce di introdurre tali modifiche *progressivamente* in modo da mantenere il sistema in condizioni tali da poter essere integrato in maniera continuativa rimanendo funzionante (e quindi in grado di fornire feedback al team).

In certi casi questo può non essere semplice (ad esempio con modifiche al database schema, che richiede modifiche significative per poter tenere il sistema funzionante). Nel qual caso è opportuno **versionare tutti i componenti interessati**, il che permette di isolare i cambiamenti e far funzionare certi componenti in parallelo finchè il nuovo non è completamente funzionante e può sostituire il vecchio.

> Keep your project releasable at all times. Ensure that the project is always compilable, runnable, tested, and ready to deploy at a moment's notice.

Qualora nemmeno questo fosse possibile, come eccezione si può rendere il codice instabile per un breve periodo di tempo (ad esempio una settimana) se il mantenerlo stabile durante la transizione è troppo oneroso. Ma avere sempre un branch stabile nel sistema di versionamento a cui poter tornare e con il quale è possibile sperimentare: non c'è alcuna ragione per rendere il sistema instabile in modo *irreversibile*.

4.4.1 What it feels like

Si è sempre in grado, in qualsiasi momento, di mostrare una versione stabile e funzionante del software a chiunque.

4.5 14. Integrate early, integrate often (INTEGRAZIONE CONTINUA)

Non attendere l'ultimo momento per integrare i componenti. Più tempo passa, più il lavoro di persone diverse può divergere e di conseguenza più difficile e onerosa sarà l'integrazione.

Invece integrare spesso e fin dall'inizio è meno stressante e aiuta a coordinare gli sforzi mantenedo i diversi moduli sincronizzati e ingrado di lavorare insieme (ovvero, il codice rimane in uno stato consistente. Vedi punto precedente).

Integrare spesso evidenzia come i vari moduli integragiscono, condividono e si scambiano informazioni. Prima si comprendono questi aspetti, meno lavoro di integrazione ci sarà da fare.

C'è da equilibrare le esigenze di *isolare* e *integrare* le modifiche. Ma l'ideale è **integrare diverse volte al giorno**, e comunque possibilmente mai andare oltre i due-tre giorni senza.

> Integrate early, integrate often. Code integration is a major source of risk. To mitigate that risk, start integration early and continue to do it regularly.

4.5.1 What it feels like

Quando fatta correttamente, l'integrazione è continua: diventa parte del processo di sviluppo e non più così onerosa. I problemi vengono individuati non appena sorgono quando sono ancora piccoli e facilmente risolvibili, senza aspettare che diventino insormontabili.

4.6 15. Automate deployment early (DEPLOYMENT AUTOMATICO)

Il fatto che il software funzioni correttamente nelle macchine utilizzate per lo sviluppo o nel server del team non significa automaticamente che funzionerà anche in quelle degli utenti.

Il prodotto deve funzionare nelle macchine di produzione, e si deve essere sicuri di poter fare il deployment nelle macchine target in modo *affidabile e ripetibile*. Allo stesso modo in cui si è automatizzata e inserita quotidianamente l'integrazione nel processo di sviluppo, lo stesso si può fare analogamente con il deployment.

Automatizzare e testare spesso il deployment aiuta a scoprire errori e difetti quando ancora sono "nuovi" (file mancanti, dipendenze non soddisfatte, struttura delle directory impropria, ecc), piuttosto che incapparci alla fine dello sviluppo o peggio dopo la consegna al cliente.

> Deploying an emergency bug fix should be easy, especially in a production server environment. You know it will happen, and you don't want to have to do it manually, under pressure, at 3:30 a.m.

Il deployment automatico aiuta a testare la procedura di installazione, ma bisogna fare attenzione a rendere esplicito nel software di che versione si tratta per la risoluzione di eventuali problemi. Per questo *il numero di versione dovrebbe essere avanzato automaticamente.*

Predisporre il testing, la continuous-integration e l'installer PRIMA ANCORA di partire con il progetto facilita molto le cose.

> Deploy your application automatically from the start. Use that deployment to install the application on arbitrary machines with different configurations to test dependencies. QA should test the deployment as well as your application.

TDD, CI e CD (continuous-deployment) sono un investimento che va fatto all'inizio, ma che ripaga abbondantemente nel corso di tutto lo sviluppo. Inoltre, una volta imparato come procedere diventa molto semplice e veloce predisporre l'infrastruttura.

4.6.1 What it feels like

L'installazione e/o il deployment del prodotto dovrebbero essere *veloce, trasparente, affidabile e ripetibile.* Funziona e basta.

4.7 16. Get frequent feedback using demos (FEEDBACK RAPIDO)

I requisiti sono qualcosa di *fluido* che varia nel tempo, e un buon modo di lavorare deve tenere conto di questo fatto.

Le idee, esigenze e prospettive cambiano con l'andare del tempo, specialmente una volta che il cliente ha qualcosa di concreto con cui testarle. E' la natura delle cose ed è necessario *adattarsi ad essa* in modo agile.

La chiave per la soddisfazione del cliente è **costruire quello che vuole, il che non è necessariamente quello che ha chiesto**. Per capire quello che realmente desidera è sottoporgli molto presto un prototipo su cui possa dare feedback riguardo la direzione da intraprendere. Lo sviluppo del software diventa uno sforzo collaborativo in cui il cliente viene coinvolto e aiuta a guidare lo sviluppo.

> It's pretty easy to see that the larger the gap between when you get their requirements and the time you show them what you've done, the further off course you'll be.

Metafora: più piccolo è l'intervallo di campionamento di un segnale, più accurata è la ricostruzione dell'informazione tramite interpolazione.

Partendo dalla consapevolezza che i requisiti *non* sono fissi e chiari fin dall'inizio e che nemmeno il cliente in genere sa con precisione di che cosa ha bisogno fin nei minimi particolari, diventa chiara l'utilità di sviluppare il software in modo incrementale e iterativo coinvolgendo maggiormente il cliente.

Questo comporta alcuni importanti vantaggi *per il cliente*:

- E' coinvolto maggiormente nel processo di sviluppo e vede con frequenza i progressi (e dove vanno i suoi soldi).

- Vedendo spesso delle demo è in grado di chiarirsi le idee e aiutare il team guidandolo nella realizzazione di un prodotto che soddisfi le sue *reali* esigenze.
- E' in grado di prioritizzare i requisiti in base al contesto attuale (progressi compiuti, tempo e soldi disponibili) e ottenere prima quello che è più importante per lui.

  Develop in plain sight. Keep your application in sight (and in the customers' mind) during development. Bring customers together and proactively seek their feedback using demos every week or two.

4.7.1 Mantenere un glossario di progetto

E' importante che tutti gli stakeholdes parlino lo stesso linguaggio.

Creare un documento online (su wiki ad esempio) a cui possano far riferimento tutti quanti e utilizzare tali termini per modellare il dominio applicativo a livello di codice (*DDD: Domain-Driven Design*).

4.7.2 Issue Tracking

Tutto questo feedback arriva sotto forma di bug, richieste di modifica, nuove feature e altro. Occorre avere predisposto un sistema di issue-tracking per gestire in modo ordinato e centralizzato questa mole di informazioni.

4.7.3 What it feels like

Dopo un periodo di rodaggio ci si cala nel ritmo, le sorprese diventano relativamente rare e il cliente ha un senso di relativo controllo sul prodotto finale.

4.8 17. Use short iterations. Release in increments (SVILUPPO ITERATIVO E INCREMENTALE)

Le metodologie agili racconmandano uno sviluppo *iterativo* e *incrementale*. Dove incrementale significa che il prodotto viene sviluppato implementandolo a piccoli gruppi di funzionalità. Iterativo vuol dire che il processo di sviluppo (analisi, progettazione, codifica, testing e analisi del feedback) è portato avanti ripetutamente nel corso di diverse rapide iterazioni.

Lo sviluppare un prodotto in corte iterazioni e in modo incrementale permette di poter presentare un demo o una release al termine di ogni iterazione. Al termine di ogni *incremento* c'è una *milestone*, ovvero una versione

rilasciabile e utilizzabile del prodotto finale. Un incremento si ha con un dopo un certo numero di iterazioni.

Sistemi grossi e complessi comportano un importante rischio a causa della fluidità dei requisiti, quindi creare un piano dettagliato a lungo termine o utilizzare iterazioni troppo lunghe è un segnale abbastanza chiaro che il progetto ha ottime probabilità di fallire. L'approccio agile di sviluppare un'applicazione a piccoli passi (iterazioni) raccogliendo fin da subito il feedback del cliente riduce considerevolmente questo rischio.

> Show me a detailed long-term plan, and I'll show you a project that's doomed.

Il modo migliore per limitare il più possibile il rischio correlato alla fluidità dei requisiti è raccogliere il feedback del cliente il prima possibile e di frequente. Per raccogliere questo feedback occorre quindi individuare il *core* di funzionalità che rendono il prodotto usabile e realizzarle il prima possibile per poterlo mettere in produzione e dare così agli utenti reali qualcosa su cui poter esprimere suggerimenti, osservazioni e requisiti migliori (su funzionalità, usabilità, ecc).

Chiedere al cliente quali sono le funzionalità essenziali (senza lasciarsi distrarre da tutte le possibilità aggiuntive o anche da una interfaccia grafica attraente), *e farle avere il prima possibile agli utenti reali in produzione.*

Durate ideali dei cicli di sviluppo:

- *Incremento*: da qualche settimana a qualche mese.
- *Iterazione*: non più di un paio di settimane, da terminare con una demo e una versione del software da mettere in mano ad alcuni utenti selezionati per ricevere feedback. La lunghezza delle iterazioni dovrebbe essere lo stesso per tutte, si deve stabilire un *ritmo* (adattare scope dei task e lunghezza delle iterazioni).

> Develop in increments. Release your product with minimal, yet usable, chunks of functionality. Within the development of each increment, use an iterative cycle of one to four weeks or so.

4.8.1 What it feels like

Si ha una breve iterazione focalizzata e con un obiettivo concreto e ben definito (che viene raggiunto). Una deadline fissa forza a prendere le decisioni importanti senza che rimangano in sospeso per troppo tempo.

4.9 18. Fixed prices are broken promises (STIME DI TEMPI E COSTO)

L'ideale sarebbe poter prevedere esattamente sia i tempi che i costi della realizzazione di un software, ma sfortunatamente questo è praticamente impossibile: si commettono errori (siamo umani), i requisiti cambiano nel tempo (e non sono tutti chiari all'inizio), le performance dei vari team non sono le stesse (ci sono differenze anche del 20X o più in base a diversi studi) e l'arrivo costante di nuove tecnologie.

> A fixed price guarantees a broken promise.

Data la non riproducibilità e la volatilità dei prodotti software, parlare di un prezzo significa fare una promessa che non verrà mantenuta nella maggior parte dei casi. Qual'è la soluzione, tenendo conto dello sviluppo interativo e incrementale? Fare delle stime migliori o prendere accordi diversi?

Se il prezzo deve essere fisso, si può fare una stima con diversi tipi di analisi (COCOMO, Fixed Point analysis, ecc). Ma le stime sono più affidabili se lo stesso team ha già affrontato un progetto simile in passato ed ha esperienza nel dominio applicativo.

Nel resto dei casi, in cui ogni prodotto fa storia a sè, il processo di sviluppo deve essere uno sforzo collaborativo di scoperta e invenzione. Perciò è opportuno negoziare un tipo diverso di accordo, per esempio:

1. Offrire di costruire un piccolo e utile sottoinsieme del sistema, scegliendo le feature in modo che non ci vogliano più di 6–8 settimane ma che si ottenga qualcosa di usabile.

2. Alla fine di questa iterazione, il cliente può provare il software e decidere se continuare con la prossima iterazione o cancellare il progetto, pagare solo per il lavoro iniziale e affidare l'incarico a qualche altro gruppo.

3. Se il cliente decide di continuare, si è in posizione migliore per poter stimare quanto può essere fatto nella prossima iterazione. Al termine di ogni iterazione il cliente ha la stessa scelta del punto 2.

> If you aren't comfortable with the answer, see if you can change the question.

I vantaggi per il cliente di questo accordo sono che:

- può vedere i progressi (o la mancanza di essi) fin da subito
- ha sempre il controllo e può staccare la spina in ogni momento senza alcuna penalità contrattuale

- ha il controllo su quali funzionalità verranno realizzate per prime e quanti soldi spenderà

In definitiva per il cliente ci sono molti meno rischi.

E per l'organizzazione e il team di sviluppo ci sono tutti i vantaggi dell'approccio iterativo e incrementale.

> Estimate based on real work. Let the team actually work on the current project, with the current client, to get realistic estimates. Give the client control over their features and budget.

In ogni caso può essere opportuno fornire una stima (per quanto irrealistica) prima di cominciare.

Una ulteriore possibilità è quella di prevedere contrattualmente un prezzo fisso per iterazione e lasciare il numero di iterazioni non precisato.

4.9.1 What it feels like

Le stime cambiano ed evolvono nel corso del progetto, ma con ogni iterazione le stime diventano sempre più precise.

Capitolo 5

Agile Feedback

Una parte importante dell'agilità è la raccolta e l'uso di feedback per fare tanti piccoli aggiustamenti (il modo principale per reagire ai cambiamenti).

Una parte dei feedback, come trattato, si ottengono direttamente dal cliente tramite uno sviluppo incrementale e iterativo, mostrando una demo o una release presto e spesso. Un'altra è assicurarsi che la base di codice sia in salute e in grado di supportare nuove features.

5.1 19. Put angels on your shoulders (AUTOMATED UNIT TESTS)

L'agilità riguarda il reagire al cambiamento, e il codice è la cosa che cambia più in fretta in un progetto. Di conseguenza è necessario ottenere feedback continuo da esso per assicurarsi che i cambiamenti siano non peggiorativi (*coding feedback*).

```
unit testing = coding feedback technique
```

Utilizzare i test di unità e *automatizzarli* in modo che vengano eseguiti:

- in locale ad ogni compilazione del codice
- remotamente da un server per la continuous-integration che scarica, compila e testa il codice costantemente comunicando immediatamente a tutto il team se qualcosa va storto

Questo è il genere di feedback continuo che ci serve. Se qualcosa non funziona lo si saprà subito, e aggiustare la situazione sarà molto meno costoso e difficile.

Per questo è utile considerare le *test-suite* come un passo aggiuntivo del processo di compilazione del codice ed eseguirla molto di frequente (quindi è importante che la suite di test esegua velocemente: più tempo ci vuole

per eseguire i test, meno è probabile che verranno eseguiti spesso come è necessario).

> Use automated unit tests. Good unit tests warn you about problems immediately. Don't make any design or code changes without solid unit tests in place.

5.1.1 Motivazioni per i test di unità

- **Forniscono feedback istantaneo**: si ha la sicurezza di poter modificare e ridisegnare il codice senza temere di introdurre errori o alterarne il *comportamento*. Si ha del feedback per evidenziare tempestivamente ogni problema e correggerlo prima che diventi troppo oneroso eliminarlo.

- **Rendono il codice robusto**: il *focus* sul comportamento aiuta a realizzare componenti robusti e ad esercitarne casi positivi, negativi ed eccezionali.

- **Possono essere un utile aiuto alla progettazione**: dato che il modo migliore per progettare un componente è capire cosa questo deve fare e come si comporta dal punto di vista di chi vi collabora, il realizzare i test *prima* del codice aiuta a chiarirne le *responsabilità* e di conseguenza a progettarlo nel modo migliore.

- **Incrementano la fiducia nel codice prodotto**: quando il codice è testato estensivamente e in una varietà di situazioni diverse, si ha maggior sicurezza nel lavorarci e modificarlo in presenza di deadline imminenti e sotto pressione.

- **Possono agire da "sonde" nella soluzione di probemi**: i test di unità possono essere utilizzati per evidenziare e risolvere problemi in qualsiasi parte del codice non appena sorgono. Se si scopre un bug che non è evidenziato dai test, scriverne subito uno o più che lo esibiscono, in modo che non riemerga più in futuro. E' come un oscilloscopio o un tester per verificare certi parametri in qualsiasi punto di un circuito, con la differenza che i controlli sono automatici e a tappeto.

- **Sono una documentazione affidabile**: esercitando il codice dal punto di vista dell'utente, possono funzionare da *documentazione eseguibile* che quindi è sempre in sincrono con il codice.

- **Sono un'aiuto nell'apprendimento**: quando si deve imparare una nuova API, è utile scrivere dei test di unità per ottenerne una documentazione accurata e affidabile.

5.1.2 Refactoring

Inoltre, i test di unità costituiscono una suite di test di regressione. Il che consente la pratica del **refactoring** con la sicurezza di poter modificare la struttura del codice senza alterarne le funzionalità.

Per refactoring si intende l'attività di modificare la *struttura* del codice senza alterarne il *comportamento*.

Il refactoring è necessario per mantenere la base di codice in salute, ordinata, espandibile e chiara. Ma non sarebbe conveniente fare queste modifiche senza la suite di test.

> With unit tests in place, acting as regression tests, you're now free to refactor the code base at will. You can rewrite and redesign code and experiment as needed: the unit tests will ensure that you haven't broken anything accidentally. That's a very powerful freedom; you don't have to code as if "walking on eggshells."

5.1.3 Commento (BDD)

La parola "test" genera confusione e rende difficile per i newbie *afferrare* il senso e l'uso corretto del TDD. Un esperimento cognitivo è quello del BDD (Behavioural-Driven Development) che si può definire in breve come "il TDD fatto correttamente".

In pratica si tratta di utilizzare un diverso vocabolario (*specificha* al posto di test ad esempio) e una prospettiva diversa che enfatizza la specifica del comportamento di un sistema (sia dal punto di vista di sistema che di unità) piuttosto che la sua struttura. Questo comporta l'utilizzare maggiormente il BDD (o TDD) per guidare il design piuttosto che per la possibilità di avere dei test di regressione (pur essendo questo un importante effetto).

L'obiettivo del BDD (cioè del TDD fatto correttamente) è quello di ottenere delle *specifiche eseguibili* che descrivono il *comporamento* di un sistema e/o di una unità in modo da guidarne la realizzazione.

Nell'ottica di specificare in isolamento il comportamento di un componente dal punto di vista delle sue interazioni con altri componenti, si utilizzano i *mock objects*.

5.1.4 What it feels like

Si diventa *test-infected*: scrivere codice senza dei test diventa come minimo "fastidioso" e decisamente non confortevole, è come camminare in bilico su una fune sospesa nel vuoto senza alcuna rete di protezione.

5.2 20. Use it before you build it (TDD)

Una larga parte dello sviluppo consiste nel realizzare API e interfacce che dovranno essere utilizzate da altri componenti o da terze parti.

Occorre *utilizzare* tali interfacce addirittura *prima di cominciare a scrivere il codice* che le implementa. Questo comporta un paio di vantaggi molto importanti:

1. Utilizzando in prima persona il codice sotto test, si guarda il componente dal punto di vista dell'utente, non di chi la implementa. Questo permette di realizzare interfacce più consistenti e usabili.

2. Partendo dai test permette di focalizzarsi sul comportamento richiesto e di realizzare tutto e solo il codice necessario ad assolvere tale comportamento. E' una spinta a *semplificare* il più possibile il design implementando il minimo codice indispensabile per esibire un certo comportamento (seguendo il principio *YAGNI*), mantenendo contemporaneamente il maggior numero possibile di strade aperte per implementare ulteriori features (usando il *refactoring*).

> Good design doesn't mean more classes.

Nel TDD è importante eseguire i test appena vengono scritti, anche quando falliranno chiaramente perchè la classe che testa non esiste ancora. A volte non falliscono come ci si aspetterebbe, il che aiuta a catturare il prima possibile eventuali errori commessi nella scrittura del test.

> Use it before you build it. Use Test Driven Development as a design tool. It will lead you to a more pragmatic and simpler design.

5.2.1 Quando non usare il TDD

Quando si vuole semplicemente esplorare nuove idee o realizzare prototipi, per loro natura si può fare a meno dell'approccio TDD (visto che in genere si usano solo come proof-of-concept, se si decide di implementarli per metterli in produzione conviene partire da zero e riscriverli).

5.2.2 What it feels like

Si sa sempre il motivo per cui si scrive del codice e si è in grado di progettare un componente concentrandosi sull'interfaccia e sul suo comportamento senza lasciarsi distrarre dai dettagli implementativi.

5.3 21. Different makes a difference (TESTARE SU DIVERSE PIATTAFORME)

Differenze nella piattaforma (hardware e software) con tutta probabilità comportano differenze nell'esecuzione.

Oltre a testare automaticamente e continuamente il codice sulla propria macchina di lavoro, è necessario testarlo anche su tutte le piattaforme che dovranno essere supportate utilizzando il server di CI: oltre a testare il codice automaticamente ad ogni push, si occuperà di testarlo su tutte le piattaforme che devono essere supportate.

> Different makes a difference. Run unit tests on each supported platform and environment combination, using continuous integration tools. Actively find problems before they find you.

Soprattutto, in generale:

> Actively find problems before they find you.

5.3.1 What it feels like

E' come il TDD, solo attraverso "mondi" diversi.

5.4 22. Automate acceptance testing (ACCEPTANCE TESTING)

La parte critica di *business logic* deve essere testata e approvata separatamente dal resto del sistema da parte dell'utente. Questo testing, può e deve essere automatizzato allo stesso modo dei test di unità.

> Create tests for core business logic. Have your customers verify these tests in isolation, and exercise them automatically as part of your general test runs.

5.4.1 What it feels like

E' come scrivere test di unità in modo collaborativo: gli sviluppatori scrivono i test, ma le risposte corrette vengono fornite dal cliente stesso.

5.5 23. Measure real progress (STIME DI TEMPO)

La misura migliore e più utile non è la percentuale di completamento di ogni task, ma *quanto manca alla sua completa realizzazione.*

Prima di cominciare la misura sarà una stima di quanto tempo si pensa occorrerà per realizzarla, ma alla fine è necessario tenere traccia del tempo effettivo per confrontarli e constatare quanto buona era la previsione iniziale (feedback). Le stime saranno altalenati, ma con il tempo e la pratica diventeranno sempre più precise: crescerà la capacità di stimare con crescente precisione lo sforzo richiesto da un determinato task.

> Focus on functionality, not the calendar.

5.5.1 Backlog

Un backlog è una lista prioritizzata contenente tutti i task che devono essere completati (quelli completati possono essere eliminati o marcati). Man mano che arrivano nuovi task possono essere aggiunti e prioritizzati.

```
backlog = priotitized open todo list
```

E' utile avere un backlog (di progetto, per iterazione, personale) in modo da avere sempre ben chiaro qual'è il prossimo passo e, con il migliorare delle capacità di stima, anche quanto ci vorrà.

> Measure how much work is left. Don't kid yourself—or your team— with irrelevant metrics. Measure the backlog of work to do.

5.5.2 What it feels like

E' una bella sensazione sapere e vedere cosa è stato fatto, cosa manca, e quanto ci vorrà. (Feedback!)

5.6 24. Listen to users (FEEDBACK DAGLI UTENTI)

Il software deve essere **usabile** e deve tenere conto delle esigenze e del background dei suoi utenti.

Quando qualcosa va storto, il software deve essere in grado di comunicare sufficienti dettagli per indicare qual'è il vero problema.

> Whether it's a bug in the product, a bug in the documentation, or a bug in our understanding of the user community, it's still the team's problem, not the user's.

Non basta raccogliere feedback dal codice, dal team e dal cliente. Bisogna ascoltare anche quello proveniente dagli utenti, quelli che useranno effettivamente il prodotto. Anche se questi utenti sono "stupidi" vanno ascoltati, perchè probabilmente in realtà si tratta di qualche mancanza nella documentazione o qualche altra supposizione errata.

> Every complaint holds a truth. Find the truth, and fix the real problem.

Se non è un bug nel codice, allora probabilmente si può sistemare la documentazione o il training.

5.6.1 What it feels like

Non si è seccati o rassegnati ai problemi stupidi degli utenti. Si è in grado di ascoltarli e arrivare al vero problema che sta al di sotto.

Capitolo 6

Agile Coding

> "Any fool can make things bigger, more complex, and more violent. It takes a touch of genius —and a lot of courage— to move in the opposite direction."—John Dryden

Un progetto software che parte da zero (un *green-field*) parte sempre con una base di codice comprensibile e facile con cui lavorare. Con l'andare del tempo però, l'entropia comincia ad intaccare il software che assomiglia sempre di più ad una "*big ball of mud*" e il mantenerlo diventa sempre di più uno sforzo erculeo.

Per evitare questo destino, il codice deve essere *mantenuto* dedicando un pò di tempo ogni giorno per tenerlo in ordine e impedire che diventi rapidamente ingestibile (*deve rimanere sempre comprensibile e modificabile*).

6.1 25. Program intently and expressively (CODICE LEGGIBILE)

> "There are two ways of creating a software design. One way is to make it so simple that there are **obviously no deficiencies**. And the other way is to make it so complicated that there are **no obvious deficiencies**."—"Hoare on Software Design" by C.A.R. Hoare

E' molto più importante che il codice sia *leggibile* che "furbo": il codice verrà letto molte più volte di quanto verrà scritto. Impegnarsi a scrivere e mantenere il codice leggibile e comprensibile (refactoring) è importante per far sì che rimanga *manutenibile e facilmente modificabile* anche a distanza di tempo e da programmatori diversi. Un codice incomprensibile è un probabile ricettacolo di bug (presenti o futuri).

> As a developer, you should always be asking yourself whether there are ways to make your code easier to understand.

Quando bisogna modificare il codice, la procedura è la seguente:

1. Comprendere il codice esistente
2. Capire quello che deve essere modificato
3. Effettuare le modifiche e testarle

Dato che la comprensione è la parte di solito più difficile, conviene mantenere sempre il codice non solo funzionante, ma anche leggibile in modo che il suo **intento** sia chiaro.

> **The PIE Principle** Code you write must clearly communicate your intent and must be expressive. By doing so, your code will be readable and understandable. Since your code is not confusing, you will also avoid some potential errors. Program Intently and Expressively.

Usare le feature del linguaggio per scrivere codice più espressivo, nomi di metodi che esplicitano l'intento, nominare i parametri in modo da aiutare i lettori a capirne la funzione e le eccezioni devono aiutare a capire cosa può andare storto durante l'esecuzione.

Il principio PIE deve essere rispettato subito e sempre, non deve essere un'aggiunta posticcia. Ristrutturazione e rifinitura continua del codice (refactoring) lungo tutto il processo di sviluppo.

I commenti sono utili per chiarire il funzionamento del codice, ma a volte sono presenti solo per compensare la poca leggibilità del codice stesso. Programmare in modo *intenzionale ed espressivo* migliora la leggibilità e manutenibilità del codice, riducendo al tempo stesso la necessità di commenti e documentazione.

> Write code to be clear, not clever. Express your intentions clearly to the reader of the code. Unreadable code isn't clever.

Inoltre, considerare tutte le assunzioni che si hanno durante lo sviluppo: ciò che è chiaro ora a noi potrebbe non esserlo per qualcun altro o a noi stessi tra qualche tempo. Pensare al codice come ad una capsula del tempo che verrà aperta in qualche momento imprecisato del futuro.

6.1.1 What it feels like

Si ha la certezza che chiunque sarà in grado di leggere il codice tra un anno e capirlo alla prima lettura.

6.2 26. Communicate in code (DOCUMENTAZIONE DEL CODICE)

E' necessario documentare il codice in due modi: attraverso il codice stesso e usando i commenti per comunicarne l'*intento*.

Il codice deve essere sufficientemente chiaro ed elegante da essere comprensibile, e i commenti vanno usati per documentare il comportamento del metodo e il suo intento (con eventualmente dettagli ad alto livello di come interagisce col resto del sistema). Evitare di inserire commenti ovvi solo perchè si ritiene di dover inserire qualcosa.

> Commenting what the code does isn't that useful; instead, comment why it does it.

Tipicamente un buon commento per un metodo contiene:

- *Intento*: cosa fa il metodo, qual'è il suo scopo
- *Pre-condizioni*
- *Post-onditioni*
- *Eccezioni*

> Comment to communicate. Document code using well chosen, meaningful names. Use comments to describe its purpose and constraints. Don't use commenting as a substitute for good code.

Il codice leggibile e i commenti che ne chiariscono comportamento, vincoli e ambiente durante l'esecuzione creano tutta la documentazione necessaria a livello di codice.

6.2.1 What it feels like

I commenti diventano utili: basta leggerli e dare un'occhiata al codice per capire che cosa fa e come.

6.3 27. Actively evaluate trade-offs (OPERARE COMPROMESSI)

Nello sviluppo del software rientrano molti parametri, tra i quali di solito ce ne sono in contrasto tra loro. Non è possibile ottimizzarli tutti.

Di solito occorre operare dei compromessi in base a quali sono i *valori* per il cliente. Sia per quanto riguarda l'intero processo di sviluppo (compromesso tra costi, tempi e completezza), sia per la realizzazione effettiva

(compromessi tra performance, eleganza, configurabilità, facilità di sviluppo, "purezza" del paradigma seguito, ecc).

Non c'è una soluzione *migliore*, ma solo quella più conveniente per la situazione in esame: consultare gli stakeholder per decidere dove concentrare gli sforzi e considerare la situazione in modo pragmatico.

> Actively evaluate trade-offs. Consider performance, convenience, productivity, cost, and time to market. If performance is adequate, then focus on improving the other factors. Don't complicate the design for the sake of perceived performance or elegance.

6.3.1 What it feels like

Non si può avere il controllo di tutto e fare ogni cosa come si vorrebbe, ma in ogni caso sono presenti le cose importati (per il cliente).

6.4 28. Code in increments (MIGLIORAMENTO CONTINUO DEL CODICE)

Così come l'approccio agile suggerisce in scala di progetto di procedere in modo incrementale e iterativo, così è utile farlo durante la codifica: bisogna fermarsi spesso per essere sicuri di procedere nella direzione giusta e per reagire tempestivamente ad ogni necessità che si presenti raccogliendo e analizzando spesso il feedback proveniente dal codice (attraverso il testing).

Programmare in piccoli incrementi tende a produrre codice più corto, leggibile e coesivo, oltre che a consentire di catturare e correggere subito gli errori. Al termine di ogni incremento poi, è utile cercare sempre dei modi per migliorare struttura e leggibilità del codice tramite il *refactoring*.

Usare il TDD per l'enforcing dell'approccio agile: fare qualcosa di breve spesso, piuttosto che aspettare di fare tutto quanto alla fine.

> Write code in short edit/build/test cycles. It's better than coding for an extended period of time. You'll create code that's clearer, simpler, and easier to maintain.

6.4.1 What it feels like

Test-infected! Si sente il bisogno (professionale) di effettuare un ciclo di build/test ogni poche righe di codice.

6.5 29. Keep it simple (SEMPLICITA')

Non iper-ingegnerizzare o iper-complicare il software. Ogni azione intrapresa nello sviluppo deve essere la soluzione ad un particolare problema immediato.

Diversamente si rischia di introdurre complessità senza che ce ne sia un reale bisogno. L'obiettivo dovrebbe essere quello di mantenere il software il più semplice possibile che sia in grado di soddisfare i requisiti (e di supportare eventuali modifiche): più il codice è piccolo e semplice, più è in grado di supportare modifiche e meno errori conterrà (maggiore manutenibilità).

> "La semplicità è la sofisticazione ultima."—Leonardo Da Vinci

Come ogni scrittore sa, anche se la complessità può impressionare la vera maestria sta nel rendere le cose difficili semplici.

> Simple is not simplistic.

Un programmatore esperto è in grado di implementare una determinata funzionalità nel modo più semplice possibile (e più manutenibile), ma per fare questo il codice attraversa molte iterazioni di refactoring per semplificarlo. Analogamente a quanto uno scrittore fa con i suoi pezzi: devono attraversare molti cicli di editing prima di assumere la forma definitiva.

> "I would have written a shorter letter, but I did not have the time."—Blaise Pascal

> Develop the simplest solution that works. Incorporate patterns, principles, and technology only if you have a compelling reason to use them.

6.5.1 What it feels like

Non ci sono più righe di codice da eliminare e il software continua a fornire tutte le funzionalità. Il codice quindi risulta più facile da leggere e correggere.

Il codice è *elegante*.

> Elegance is easy to understand and recognize but much harder to create.

6.6 30. Write cohesive code (SRP)

La *coesione* è una caratteristica di un componente che ne descrive quanto le funzionalità dei suoi membri sono correlate. In altre parole, un componente con alta coesione avrà dei membri le cui funzionalità riguardano una sola o un ristretto gruppo di features. Conosciuto anche come *SRP (Single Responsibility Principle)*.

Questo modo di organizzare i membri (ad esempio metodi) raggruppandoli per funzionalità in moduli diversi consente di ottenere una migliore

produttività e *manutenibilità*. Se un componente avesse poca coesione e contenesse più feature, le possibilità che possa essere modificata sono più alte ed essendo un componente importante le modifiche potrebbero propagarsi ad altri componenti rendendo rapidamente qualsiasi modifica potenzialmente molto onerosa.

Conseguenze ci sono anche per quanto riguarda la *riusabilità*: più un componente o libreria contiene funzionalità scorrelate, meno è riutilizzabile.

> Keep classes focused and components small. Avoid the temptation to build large classes or components or miscellaneous catchall classes.

Ma c'è un limite alla separazione di responsabilità, oltre al quale la coesione diventa più un'onere che un beneficio: bisogna evitare di arrivare ad uno "spaghetti OOs" system.

6.6.1 What it feels like

Le classi e le componenti sono ben focalizzate: fanno una cosa sola e la fanno bene. E' più facile rintracciare i bug ed effettuare modifiche perchè le responsabilità sono ben definite.

6.7 31. Tell, don't ask (INCAPSULAZIONE)

> "Procedural code gets information and then makes decisions. Object-oriented code tells objects to do things."—Alec Sharp

A differenza del codice procedurale, un sistema creato secondo il paradigma OO è composto da oggetti (con alta coesione) i quali forniscono le funzionalità richieste comunicando tra loro e scambiandosi *messaggi*.

Il principio dell'*incapsulazione* (di dati e funzionalità) prescrive che ogni oggetto implementi la propria logica (non modificabile direttamente da altri) e non modifichi lo stato di altri oggetti. In altre parole *un oggetto è un agente in un sistema complesso*. Di conseguenza quando un oggetto richiede una funzionalità offerta da un altro componente deve *chiedere* a tale oggetto di fornirla, non recuperarne lo stato ed effettuare il calcolo personalmente (sarebbe una violazione del principio di incapsulazione).

Prendere decisioni al di fuori della sfera di influenza di un oggetto è uno dei modi più rapidi per produrre codice poco manutemibile e ricco di bug.

> Tell, don't ask. Don't take on another object's or component's job. Tell it what to do, and stick to your own job.

6.7.1 Command-query separation

Un utile corollario della regola "tell, don't ask" è quella di categorizzare esplicitamente (e documentarli come tali) funzioni e metodi come:

- **Comando**: un messaggio che inviato ad un altro oggetto ne implica una modifica dello stato ed eventualmente ottiene indietro dei risultati (implica side-effects)
- **Query**: un messaggio che ottiene indietro dei valori senza modificare lo stato dell'oggetto a cui la invia (nessun side-effect)

Separare e identificare esplicitamente comandi da query incoraggia l'uso dell'incapsulazione. Inoltre con le query esplicitamente segnalate, ci si può chiedere perchè una determinata informazione è fornita all'esterno ed eventualmente se è il caso di convertirla in un comando.

6.7.2 What it feels like

Rispettare il principio di incapsulazione significa poter vedere le interazione tra oggetti come *passaggio di messaggi* non come chiamate di metodi. E' come inviare messaggi piuttosto che eseguire funzioni (allontanarsi dal paradigma procedurale).

6.8 32. Substitute by contract (LSP, FLESSIBILITA')

Il modo più comune per rendere un sistema *flessibile* è permettere di poter sostituire delle parti di codice senza che il resto del software debba essere modificato.

Per ottenere questa proprietà occorre rispettare il *Liskov's Substitution Principle*:

> Any derived class object must be substitutable wherever a base class object is used, without the need for the user to know the difference.

Se nell'ereditarietà tale principio non viene rispettato è comunque possibile ottenere delle gerarchie di classi che forniscono la *riusabilità*, ma senza la *flessibilità*: il chiamante deve sapere il tipo esatto dell'oggetto per sapere come gestirlo.

> Inheritance isn't evil, just misunderstood.

Un oggetto derivato, per rispettare il LSP, deve **richiedere non di più e fornire non di meno** della sua classe base.

In generale, l'ereditarietà viene abusata. Molto più spesso che no, **nel dubbio è meglio usare la *composizione* (delegazione) e riservare l'ereditarietà per quando serve davvero**.

> Delegation is usually more flexible and adaptable than inheritance.
>
> Extend systems by substituting code. Add and enhance features by substituting classes that honor the interface contract. Delegation is almost always preferable to inheritance.

6.8.1 What it feels like

Il sistema è molto flessibile: è possibile implementare e inserire una nuova classe per aumentare o modificare le funzionalità del software senza dover modificare il resto del sistema.

Capitolo 7

Agile Debugging

Non è possibile produrre subito qualcosa di una certa complessità senza compiere nessun errore. Si possono adottare pratiche e processi che minimizzano il numero di questi errori e permettono di individuarli e correggerli presto quando ancora il costo è basso, ma ne verranno sempre commessi.

Sfortunatamente, a differenza dei design-meetings il *debugging* non può essere sottoposto a time-boxing.

7.1 33. Keep a solution log (DAYLOGS)

E' inutile reinventare la ruota e sprecare tempo cercando la soluzione agli gli stessi problemi (o problemi simili) ancora e ancora (tremendamente frustrante), quando quel tempo potrebbe essere utilizzato più proficuamente risolvendone di nuovi.

> Don't get burned twice.

Cercare la risposta su internet può aiutare, ed è certo meglio che sforzarsi in isolamento (per la seconda volta almeno sullo stesso problema), ma non sempre si trova la soluzione e può volerci molto tempo.

Un modo più furbo per gestire la faccenda è annotare ogni *problema* incontrato e la relativa *soluzione* trovata in un **daylog**: in questo modo quando si reincontra un problema già risolto in passato basta cercare in tale log la soluzione trovata senza dover sprecare tempo a ricercarla nuovamente da zero.

Alcune informazione che potrebbero essere utili per ogni entry di questo log:

- Data del problema
- Breve descrizione del problema
- Versione di applicazioni, framework e sistemi

- Descrizione dettagliata della soluzione (e del *perchè* è stata scelta)
- Riferimenti ad articoli e URL che contengono ulteriori dettagli o informazioni correlate
- Ogni frammento di codice, settaggio e snapshot di finestre che potrebbero essere parte della soluzione o aiutare nel comprendere più a fondo i dettagli

Questo log dovrebbe essere facilmente editabile e indicizzato per parole chiave in modo da trovare rapidamente le soluzioni.

Ancora meglio, è avere un log *condiviso* da tutto il team (e meglio ancora, tutta l'organizzazione).

> Maintain a log of problems and their solutions. Part of fixing a problem is retaining details of the solution so you can find and apply it later.

7.1.1 What it feels like

E' come un'estensione del proprio cervello che aiuta a trovare dettagli di particolari problemi o problemi simili già incontrati e risolti.

7.2 34. Warnings are really errors (WARNINGS)

I warning a differenza degli errori non fanno fallire la compilazione e possono essere ignorati, ma **alcuni di questi warning possono nascondere problemi seri** che aspettano solo di saltare fuori nel momento meno opportuno (in produzione).

> Just because your compiler treats warnings lightly doesn't mean you should.

Ad esempio, del codice del genere:

```
if (theTextBox.Visible = true) {
    ...
}
```

Anche se si tratta chiaramente di un errore serio, il compilatore può produrre solo un warning:

```
Assignment in conditional expression is always constant;
did you mean to use == instead of = ?
```

Ignorare i warning è una pratica poco professionale e piuttosto *rischiosa*, perciò è meglio non farlo. Una pratica utie è quella di **istruire il compilatore a trattare i warning come errori**.

> Treat warnings as errors. Checking in code with warnings is just as bad as checking in code with errors or code that fails its tests. No checked-in code should produce any warnings from the build tools.

7.2.1 What it feels like

I warning sono avvertimenti (potenziali problemi) a cui bisogna prestare la dovuta attenzione.

7.3 35. Attack problems in isolation (ISOLAMENTO DEI PROBLEMI)

Cercare di trovare un problema nell'intero sistema è poco efficiente e scarsamente efficace. Conviene isolare l'area in cui si verifica: per fare questo il sistema deve essere adeguatamente modularizzato e ogni componente testato in isolamento prima di passare ai test di sistema.

Un grandissimo vantaggio nell'usare il TDD è che questo forza a disaccoppiare i vari componenti nella loro realizzazione per poterli testare in isolamento.

Testando continuamente i singoli componenti in isolamento e in modo automatico, quando ci sono dei problemi è immediato vedere *dove* sono localizzati (sia perchè i test riguardano un singolo componente, sia perchè eseguendoli di continuo il codice tra cui cercare quello che provoca l'errore è molto poco).

Nel caso non si dispongano di test di unità o non sia così semplice individuare il problema, occorre estrarre il modulo problematico e lavorarci in isolamento. Una buona tecnica è quella di *costruire un prototipo* contenete il modulo incriminato e semplificarlo fino a che contiene solo il codice che crea problemi. E' come estrarre il motore di un aereo mentre questo è a terra e lavorare su quello in isolamento piuttosto che in volo.

> Attack problems in isolation. Separate a problem area from its surroundings when working on it, especially in a large application.

7.3.1 What it feels like

Quando ci si trova a dover isolare un problema si arriva a dover cercare un ago in una tazzina da the, non in un pagliaio.

7.4 36. Report all exceptions (GESTIONE DELLE ECCEZIONI)

Se del codice non riesce a portare a termine il compito che gli è stato chiesto e si verificano delle eccezioni, l'utente di tale codice è utile che abbia *tutte le informazioni necessarie sul problema che si è verificato* per diagnosticare l'errore e trovare soluzioni.

Del codice può generare eccezioni e catturarle o propagarle al chiamante. Un semplice criterio da adottare è questo: **se il codice che riceve l'eccezione può effettuare il *recovering* e continuare come se niente fosse, può catturare l'eccezione. Altrimenti la deve propagare al chiamante per fargli sapere esattamente cosa è andato storto.**

> Determining who is responsible for handling an exception is part of design.

Solo così gli utenti del codice sono in grado di capire esattamente cosa è successo, o almeno di comunicarlo a chi sa capirlo e porvi rimedio.

> Handle or propagate all exceptions. Don't suppress them, even temporarily. Write your code with the expectation that things will fail.

Evitare di avere dei *catch-all* vuoti!

Inoltre, se un metodo può generare troppe eccezioni, vuol dire che ha troppe *responsabilità*: errore di design, quindi refactoring!

7.4.1 What it feels like

Si è sicuri che quando si verifica un problema verrà generata e **gestita o riportata** un'eccezione. Non ci sono blocchi catch vuoti.

7.5 37. Provide useful error messages (ERROR REPORTING)

In un'applicazione, è normale che di tanto in tanto ci sia qualche malfunzionamento. Ci possono essere tanti motivi: dipendenze mancanti, errori di utilizzo, ecc. Quello che bisogna fare è *registrare* e *riportare* questi errori in modo sufficientemente chiaro.

Non basta un "errore generico", servono dei dettagli per far capire all'utente qual'è il problema e cosa è possibile fare per porvi rimedio, o almeno che sia in grado di fornire abbastanza dettagli al supporto tecnico per diagnosticare e risolvere il problema.

Un primo passo per tenere traccia degli errori è il *logging*, ma deve essere analizzato dagli sviluppatori e non è di nessun aiuto all'utente.

La soluzione è tenere traccia degli errori (eccezioni) e comunicarle all'utente. Tuttavia non è bene confonderlo con messaggi troppo dettagliati, quelli stessi dettagli però che possono fare la differenza per gli sviluppatori tra una frustrante ricerca e una veloce soluzione. Invece occorre fornire:

- un messaggio di alto livello che spiega a grandi linee l'errore e può essere sufficiente all'utente per tentare un workaround
- un messaggio contenente tutti i dettagli di basso livello tali da permettere agli sviluppatori di capire subito l'errore e correggerlo

```
Couldn't login [details...]
```

Durante lo sviluppo si possono comunicare direttamente i messaggi di errore più dettagliati. Per la versione da rilasciare si può collegare attraverso hyperlink il messaggio di alto livello ad un handle di un'entry del log (o si può addirittura prevedere di inviare automaticamente un'email al supporto tecnico con tutti i dettagli rilevanti del log e uno snapshot dello stato del sistema nel momento in cui si è verificato l'errore).

> Debugging information is precious and hard to come by. Don't throw it away.

Il sistema di error-reporting ha un grande impatto sulla produttività e sui costi di supporto perchè permette di trovare e risolvere più velocemente gli errori (sia da parte dell'utente che del supporto tecnico).

> Present useful error messages. Provide an easy way to find the details of errors. Present as much supporting detail as you can about a problem when it occurs, but don't bury the user with it.

7.5.1 Categorie di errori

Categorizzare gli errori permette di fornire informazioni più utili e mirate all'audience interessata. Le categorie principali in cui si possono catalogare gli errori sono:

- **Difetti del software**: veri bug (puntatori nulli, chiavi mancanti, ecc) L'utente o l'amministratore non ci possono fare nulla.
- **Problemi d'ambiente**: non riguardano strettamente il software ma l'ambiente di esecuzione (errori di connessione al database o alla rete, permessi insufficienti, spazio su disco insufficiente, ecc). Sono problemi su cui gli sviluppatori non hanno alcun potere ma che possono essere risolti dall'utente o l'amministratore quando vengono forniti sufficienti dettagli.

- **Errori d'utente**: dovuti ad un uso scorretto del software da parte dell'utente. Egli o l'amministratore devono solo riprovare, una volta che gli è stato detto cosa hanno sbagliato.

7.5.2 What it feels like

I messaggi di errore sono utili: si è in grado di individuare immediatamente e con precisione il problema e dove si è verificato.

Capitolo 8

Agile Collaboration

Lo sviluppo di un software moderno è uno sforzo collaborativo, quindi il successo di un progetto dipende da quanto efficacemente le persone lavorano bene in team, come interagiscono e come gestiscono le proprie attività.

L'agile riconosce un posto di primo piano alla *comunicazione*, quindi alla collaborazione (vedi *The Agile Manifesto*).

8.1 38. Schedule regulare face time (STAND-UP MEETINGS)

Per il successo di un progetto la comunicazione è essenziale: non sono con il cliente, ma anche tra gli sviluppatori per rimanere al corrente dello stato di avanzamento del progetto.

Gli stand-up meetings devono essere corti, e per impedire che durino troppo vengono fatti *in piedi*. La durata ottimale è tra 10–15 minuti a un massimo di 30.

Schedulato ogni mattina, viene dato un tempo massimo per ogni persona (circa due minuti), durante i quali risponderà alle seguenti domande per mantenere l'intervento focalizzato:

- Che cosa ho fatto ieri?
- Cosa pianifico di fare oggi?
- Quali difficoltà ho incontrato?

Per team piccoli fare uno stand-up meeting ogni giorno è troppo, si può pensare di farne un giorno si e uno no o un paio di volte a settimana.

Al meeting partecipano e hanno parola solo i componenti del team (sviluppatori e cliente). E' utile avere un *coordinatore* (può essere uno dei componenti del team a rotazione) che si assicura che i tempi e il focus sia rispettato, inoltre può annotare le *issues* che possono emergere. Se c'è bisogno di discussioni più approfondite le si fa dopo e al di fuori del meeting.

I vantaggi degli stand-up meetings sono molteplici:

- Aiutano a partire focalizzati sull'obiettivo del giorno
- Se una parsona ha un problema, ha occasione di portarla all'attenzione del team e ricevere aiuto
- Aiutano ad avere un'idea dello stato di avanzamento del progetto
- Evidenziano delle eventuali aree in cui ci sono difficoltà e potrebbero essere necessarie teste aggiuntive
- Accelerano lo sviluppo promuovendo la condivisione di codice e idee
- Incoraggiano la produttività e motivazione: vedere gli altri che riportano dei progressi motiva a fare lo stesso

  Use stand-up meetings. Stand-up meetings keep the team on the same page. Keep the meeting short, focused, and intense.

8.1.1 What it feels like

I meeting risultano utili: si ha un'idea di quello su cui ognuno sta lavorando e i problemi emergono facilmente.

8.2 39. Architects must write code (DESIGN IS DEVELOPMENT)

La comprensione di un problema e del suo *contesto* migliora con il tempo e con il feedback che si riceve dalla sua implementazione. Ecco perchè i BDUF nella maggior parte dei casi sono quasi inutili e dannosi se fatti rispettare in ogni caso.

Per la progettazione è necessario prendere sia decisioni strategiche sia decisioni tattiche (per le quali è necessaria la conoscenza e il feedback dei dettagli implementativi).

La metafora della guerra aiuta a capire il concetto: la *strategia* è il dispiegamento di truppe e risorse ad alto livello per la guerra (design architetturale), mentre la *tattica* è il piano dettagliato per ogni battaglia che è possibile fare solo conoscendo bene il contesto: terreno, truppe nemiche, ecc (design specifico, pianificabile sono al momento e con una chiara comprensione della situazione).

Inoltre, dato che il cambiamento è intrinseco al mondo del software (e a tutto il resto in realtà) occorre *pianificare anche per il mutamento*, prendendo decisioni che lascino il massimo numero possibile di scelte aperte. Ogni decisione dovrebbe essere **reversibile**.

> "The designer of a new kind of system must participate fully in the implementation."—Donald E. Knuth, "Designer of a new system"

Alla luce di queste considerazione, uno o più sviluppatori dovrebbero farsi carico della progettazione del sistema. Sarebbe bene che nessuno progettasse in isolamento, ma prendesse suggerimenti.

> A programmer who refuses to design is a person who refuses to think.

E' importante che un programmatore possa essere *anche* un progettista, non viceversa.

> Good design evolves from active programmers. Real insight comes from active coding. Don't use architects who don't code—they can't design without knowing the realities of your system.

8.2.1 What it feels like

Data la natura del processo di sviluppo (evolutiva), le sue varie attività (design, codifica, testing, ecc) sono tutte facce della stessa medaglia: *sviluppo*.

8.3 40. Practice collective ownership (COLLECTIVE CODE OWNERSHIP)

Dato che lo sviluppo di software è uno sforzo collaborativo, non ha molto senso promuovere regole di "territorialità" riguardo il codice.

L'implementare soluzioni e realizzare un'applicazione che soddisfa le esigenze del cliente è più importante di decidere chi ha le idee migliori o il diritto di modificare quale parte del codice (sempre focus sull'obiettivo).

Se un solo sviluppatore si fa carico di un sotto-sistema ed è l'unico a poterci mettere le mani:

- Tutte le modifiche a quel modulo dovranno passare da lui (collo di bottiglia) e quindi non avrà alcun incentivo ad adottare buone pratiche di codifica (ci potranno essere molti *hack*, duplicazione, accoppiamento, ecc).

- Essendo l'unico specializzato in quella parte di codice, se per qualsiasi motivo non è disponibile (in vacanza o in un'altra azienda) nessun altro saprà subentrare in tempi brevi.

Invece, chiunque dovrebbe poter lavorare con le parti di codice che comprende: correggere bug, migliorarle e renderle più chiare. Quando i programmatori lavorano a rotazione su parti diverse della stessa applicazione, ognuno potrà:

- Formarsi un'idea abbastanza precisa del funzionamento dell'intero software (architettura)
- Controllare che il codice scritto da qualcun altro sia ben fatto (*code review*)
- Proporre soluzioni e punti di vista alternativi
- Individuare problemi.
- Migliorare il modulo: bugfix, rimuovere duplicazione, rendere più leggibile, ecc.

> "We can't solve problems by using the same kind of thinking we used when we created them."—Albert Einstein

Inoltre, sapere che altre persone lavoreranno con il proprio codice spinge a essere più disciplinati e mantenerlo elegante, leggibile e flessibile.

E' vero che quando un programmatore lavora solo sul "suo" codice sarà altamente produttivo e lo sviluppo procederà più velocemente. Ma a lungo termine, **quando più occhi osservano lo stesso codice si ottiene un software di migliore qualità, più facile da manutenere e capire, e con un numero minore di errori**.

> Emphasize collective ownership of code. Rotate developers across different modules and tasks in different areas of the system.

```
Collective code ownership = Sharing knowledge, Shared Expertise
```

Naturalmente bisogna essere pragmatici e tenere da conto il contesto: se alcuni componenti richiedono una conoscenza specializzata è bene che sia sempre lo stesso programmatore ad occuparsene. Inoltre, la rotazione non è una licenza per fare "hacking"!

8.3.1 What it feels like

Ognuno è a proprio agio nel lavorare a quasi ogni parte del progetto.

8.4 41. Be a mentor (DIFFONDERE LA CONOSCENZA)

Il modo migliore per far crescere e sviluppare la conoscenza è condividerla, prima di tutto con il proprio team di sviluppo.

Quando una persona tiene quello che sa per sè senza condividerlo, sfruttandolo per atteggiarsi a superiore, non c'è alcun progresso o vantaggio (nè per il team, nè a lungo termine per la persona stessa).

La conoscenza non è come i soldi: dare dei soldi a qualcun altro significa che io ne avrò di meno dopo. La conoscenza invece è come il fuoco di una candela: aiutare ad accendere la tua non mi fa rimanere nell'oscurità. Il fuoco della conoscenza si può diffondere senza diminuire la propria intensità.

> "And no matter how many people share it, the idea is not diminished. When I hear your idea, I gain knowledge without diminishing given anything of yours. In the same way, if you use your candle to light mine, I get light without darkening you. Like fire, ideas can encompass the globe without lessening their density."—Thomas Jefferson

Arricchire il team facendo circolare le idee e le conoscenze comporta molti vantaggi a lungo termine per tutti:

- Spiegare qualcosa a qualcunaltro contribuisce a chiarire e fissare le idee
- Si possono ottenere prospettive diverse quando qualcuno fa delle domande. Si può espandere la propria conoscenza e visione comprendendo le riflessioni e i punti di vista degli altri. *Intrattenere pensieri e prospettive di altri aiuta a espandere le proprie.*
- Si motiva i propri team-mates a fare altrettanto, aumentando così il livello di competenza dell'intero gruppo
- Le domande degli altri possono evidenziare delle aree che non si conoscono o non si sono capite a sufficienza (*annotare queste aree nel daylog personale*)

> A good mentor takes notes while offering advice to others.

Quando ognuno si assume anche il ruolo di mentore, il livello di competenza dell'intero team si innalza.

Essere un mentore però non significa per forza tenere lezioni con slide o fare quiz, piuttosto vuol dire presentare le proprie nuove conoscenze a qualche *brown-bag session* e in generale aiutare il resto del team a migliorarsi mentre si aiuta sè stessi.

> Being a mentor means sharing—not hoarding—what you know. It means taking an interest in seeing others learn and develop and adding increasing value to the team. It's all about building up your teammates and yourself, not about tearing down.

Anche scrivere su un blog è utile per condividere conoscenze e ottenere almeno una parte dei vantaggi della condivisione (in questo caso per ottenerli tutti occorre che il blog sia sufficientemente popolare e frequentato da programmatori di un certo livello).

> Be a mentor. There's fun in sharing what you know—you gain as you give. You motivate others to achieve better results. You improve the overall competence of your team.

Se ci si trova a rispiegare più volte uno stesso argomento a più persone, è una buona idea prenderne nota per scrivere un articolo o un libro in materia.

La pratica del *pair-programming* è un efficace mezzo per il mentoring (oltre a tutti gli altri vantaggi): aiuta a diffondere conoscenze, con il valore aggiunto che è possibile osservare passo-passo un esperto applicarla ed essere seguito nell'applicazione in prima persona.

8.4.1 What it feels like

Insegnare è un buon modo per migliorare il proprio apprendimento. Il resto del team si sente a proprio agio nel farsi aiutare e nel fare domande.

8.5 42. Allow people to figure it out (MENTORING)

> "Dai un pesce ad un uomo e lo nutrirai per un giorno, insegnagli a pesare e lo nutrirai per una vita intera."

Diffondere la conoscenza non significa dare frettolosamente delle soluzioni e ritornare al proprio lavoro, in questo modo la gente può imparare solo a dipendere da qualcun altro che gli fornisca le risposte.

Un approccio migliore è quello di **indicare la strada e lasciare che sia il programmatore a trovare da solo la risposta**. In questo modo imparerà molto di più perchè la conoscenza sarà frutto di uno sforzo personale, non un qualcosa di imposto dall'esterno.

> [...] instead of answering something obvious like "42", ask your teammate, "Did you look at the interaction between the transaction manager and the application lock handler?"

Questo approccio porta tutta una serie di vantaggi:

- Aiuta gli altri ad imparare come approcciare un problema
- Imparano molto di più che la semplice risposta: capiscono come trovare da soli le risposte che cercano
- Non ritorneranno ancora e ancora con domande simili
- Aiuta a fare in modo che possano trovare da soli la strada quando non si è disponibili
- Possono tornare con soluzioni o idee che non si aveva considerato. Ognuno impara qualcosa e la conoscenza si accresce.

Se lo sviluppatore non trova la risposta gli si possono fornire altri indizi (domande) o, se ci sta dietro da troppo tempo, direttamente la soluzione (spiegando anche *perchè* è la soluzione giusta), senza però giudicare o farlo sentire inferiore. E' utile comunque avere un tempo massimo oltre il quale gli si fornisce la risposta: l'esperienza educativa non deve diventare una fonte di stress. Se invece torna con una risposta diversa lo si può aiutare a valutarne i pro e i contro. E se la soluzione è addirittura migliore, tutti avranno imparato qualcosa (mentore compreso).

Ogni mentore è sua volta uno "studente" (*beginner's mind*).

> As a mentor, you lead others toward solutions, motivating them to solve problems and giving them an opportunity to think and learn problem solving.

Ovvero: *"Come mentore, guidi gli altri verso le soluzioni, motivandoli a risolvere i problemi e dandogli l'opportunità di pensare e imparare l'arte del problem-solving."*

> Give others a chance to solve problems. Point them in the right direction instead of handing them solutions. Everyone can learn something in the process.

8.5.1 What it feels like

Aiutare gli altri componenti del team senza dargli la pappa pronta, guidandoli affinchè possano trovare la risposta da soli ed espandere la propria conoscenza ed esperienza nel frattempo.

8.6 43. Share code only when ready (VCS)

Peggio che non usare un VCS (Version Control System, a volte chiamato SCM: Source Code Management) è usarlo male.

E' di prioritaria importanza condividere il codice spesso (più tempo si aspetta a condividere le modifiche, più difficile sarà l'integrazione) e che sia funzionante. *Mai immettere codice non funzionante nel repository di riferimento* perchè ha conseguenze negative importanti su tutto il resto del team.

Se si utilizza un VCS decentralizzato si possono fare dei *commit* locali, ma in genere le linee guida principali sono:

- Un commit ad ogni *completamento* di un task: non inserire nel repository di riferimento codice incompleto
- Il codice "committato" deve essere funzionante (tutti i test di unità devono passare), un sistema di CI aiuta con un *enforcing* della policy
- Inviare il codice nel repository di riferimento e integrarlo spesso (più volte al giorno)

  > Share code only when ready. Never check in code that's not ready for others. Deliberately checking in code that doesn't compile or pass its unit tests should be considered an act of criminal project negligence.

8.6.1 What it feels like

Si è consapevoli che l'intero team siede dall'altro capo del sistema di versionamento, e che non appena si fa il *push* delle modifiche tutto il mondo ce l'ha.

8.7 44. Review Code (CODE REVIEWS)

E' meglio trovare i problemi e i bug nel codice non appena questo viene scritto, aspettare non farà andare via gli errori di programmazione da soli.

> If you let code sit and rot for a while, it won't smell any prettier.

Il modo più efficace per risolvere molti problemi nel codice è l'*analisi statica* formale:

> **Code Reviews and Defect Removal** by Capers Jones in Estimating Software Costs [Jon98]
>
> Formal code inspections are about twice as efficient as any known form of testing in finding deep and obscure programming bugs and are the only known method to top 80 percent in defect-removal efficiency.

Impiegare del tempo nell'analisi statica del codice, anche sotto pressione di scadenze imminenti, è tempo ben speso.

Naturalmente è importante che a revisionare il codice NON sia la stessa persona che lo ha scritto, e che lo stesso codice non sia revisionato più volte dalla stesso programmatore. Questo è un incentivo ulteriore per ogni sviluppatore a scrivere il miglior codice possibile in modo che sia leggibile, elegante e funzionante (altri programmatori leggeranno quanto ho scritto).

Di norma chi revisiona il codice dispone di una *lista di controllo* contenente i particolari da controllare (ad esempio: tutti i gestori di eccezioni sono non vuoti? Tutte le operazioni su database sono eseguite all'interno di transazioni? ecc). Un elenco minimale potrebbe essere:

- Posso leggere e comprendere il codice?
- Ci sono errori ovvi?
- Il codice avrà qualche effetto indesiderato sulle altre parti dell'applicazione?
- C'è del codice duplicato (nella stessa sezione o in altre parti del software)?
- Ci sono dei miglioramenti o dei refactoring ovvi che è possibile fare per migliorarlo?

  > Review all code. Code reviews are invaluable in improving the quality of the code and keeping the error rate low. If done correctly, reviews can be practical and effective. Review code after each task, using different developers.

Sebbene sia possibile pianificare a piacere delle sessioni di revisione del codice, l'approccio agile suggerisce di distribuire queste revisioni durante tutto il processo di sviluppo (approccio "poco&spesso", in modo incrementale e iterativo). Gli stili principali:

8.7.1 The Pick-Up Game

Non appena il codice per un determinato task è stato scritto, testato ed è pronto per essere immesso nel sistema di versionamento, viene preso e revisionato da un altro sviluppatore. E' una tecnica molto efficace.

Queste *commit reviews* sono revisioni veloci e informali e sono utili a controllare che il codice sia pronto (abbastanza buono) per essere inserito nel ramo principale di sviluppo.

E' utile ruotare le persone che revisionano il codice di un certo sviluppatore.

8.7.2 Pair Programming

E' una tecnica popolarizzata dall'*Extreme Programming* e prevede che due persone lavorino sullo stesso computer: una scrive codice (*driver*) e la seconda (*navigator*) revisiona il codice in real-time. A intervalli regolari, i ruoli si scambiano. Le coppie non sono fisse e si formano giorno dopo giorno a seconda del task e delle disponibilità.

Il pair programming permette quella che si può definire **continuous code review**.

Un valore aggiunto di questa pratica è che promuove nel modo più efficace lo scambio di conoscenze e il mentoring. In effetti è uno dei metodi migliori per addestrare un nuovo sviluppatore a lavorare nel team, insegnandogli i processi e le tecniche che dovrà adottare.

8.7.3 What it feels like

Le revisioni del codice si svolgono poco alla volta e continuamente, lungo tutto il processo di sviluppo. Sono una parte integrante del progetto, non una grande, spaventosa e logorante attività che si verifica una-tantum.

8.8 45. Keep others informed (COMUNICARE I PROGRESSI)

E' importante che il team e i manager siano informati dei progressi e delle difficoltà di tutti. Informare regolarmente tutti quanti di come va il proprio lavoro permette di avere un'idea dello stato di avanzamento e consente al team di organizzarsi in caso di difficoltà (ricevere aiuto, assegnare più persone, rimandare la feature alla prossima iterazione, ecc).

Aspettare fino all'ultimo momento per far sapere che si hanno delle difficoltà non è solo poco professionale, ma mette in difficoltà l'intero team.

Il modo migliore per condividere il proprio stato è utilizzando degli **information radiators**, ovvero mezzi di comunicazione che siano visibili da chiunque e presentino informazioni che cambiano nel tempo. Ad esempio: blog, wiki, feed RSS, Twitter/status.net, ecc. L'importante è che siano abbastanza comodi da poter essere visitati *regolarmente* da ogni componente del team.

> Keep others informed. Publish your status, your ideas and the neat things you're looking at. Don't wait for others to ask you the status of your work.

8.8.1 What it feels like

Non si viene continuamente interpellati per avere aggiornamenti sui propri progressi.

Capitolo 9

Epilogue: Moving To Agility

Basta una piccola cosa per gettare luce sull'ignoranza e dare il via ad un cambiamento in positivo: gettare un piccolo sasso in un lago forma delle onde concentriche che si allargano sempre di più.

> "As one lamp serves to dispel a thousand years of darkness, so one flash of wisdom destroys ten thousand years of ignorance."— Hui-Neng

9.1 Just one new practice

Introdurre qualche pratica di quelle trattate può servire per rendere più agili e fare la differenza tra un fallimento e un progetto di successo. Basta intervistare il team e chiedere ad ognuno:

- su cosa sta lavorando
- che cosa funziona nel processo di sviluppo
- che cosa non funziona

In base alla reali difficoltà, introdurre le pratiche che la risolvono.

9.2 Rescuing a failing project

Introdurre tutte le pratiche è meglio che introdurne solo alcune, infatti concorrono tutte a migliorare l'agilità e il modo di lavorare del team al fine di produrre prodotti di alta qualità con meno risorse (tempo, costi, fatica). Ma quando si parla di un progetto già avviato e in difficoltà, *introdurle tutte in una volta può essere dannoso.*

Quando un paziente presenta problemi di salute ed è in pericolo di vita, occorre prima stabilizzarlo e solo dopo gli si può consigliare un regime di

vita più salutare. Non si può raccomandare ad un cardiopatico che sta per avere un infarto di fare più moto e mangiare molta frutta e verdura...

Allo stesso modo, un progetto in difficoltà che rischia di fallire deve prima essere stabilizzato introducendo le pratiche di base più urgenti, solo dopo si può introdurre il resto.

9.3 Introducing Agility: the manager's guide

Procedere con calma e assicurarsi di comunicare e spiegare al team la *visione d'insieme*: che cos'è l'agilità e tenere da conto che in ultima analisi *lo sviluppo agile è teso a rendere le cose più semplici per gli sviluppatori.* Se invece di semplificare la vita del team le pratiche la complicano c'è qualcosa di sbagliato.

Le prime pratiche da introdurre:

- **Non-tecniche**
  1. Stand-up meetings: per promuovere la comunicazione e sincronizzare gli sforzi di sviluppo (auto-organizzazione e sincronizzazione in un CAS)
  2. Far partecipare attivamente i progettisti nello sviluppo
  3. Informal code reviews
  4. Coinvolgere il cliente nel processo di sviluppo
- **Tecniche**
  1. VCS
  2. TDD
  3. Build automation (continuous-integration)

Una volta gettate le *basi*, si può procedere introducendo le altre pratiche.

Molto importante è tenere delle **project retrospectives** (retrospettive) al termine di ogni iterazione o release, chiedendo a ogni componente del team:

- cosa ha funzionato
- cosa ha bisogno di aggiustamenti
- che cosa non funziona

Questo **è molto importante per il miglioramento di processo e l'adattamento al contesto quale è il team di sviluppo ed il progetto a cui si lavora.**

9.4 Introducing Agility: the programmer's guide

Per una transizione il più semplice e veloce possibile occorre il supporto del management adottando una strategia come sopra delineata. Quando ciò non fosse possibile, i programmatori che vogliono introdurre le pratiche agili devono *dare l'esempio* dato che non possono procedere per decreto.

Invece di cercare di "vendere" le pratiche al resto del team, adottarle in prima persona e lasciare che sia la migliore qualità del proprio lavoro a incuriosire gli altri e spingerli a chiedere di cosa si tratta.

Iniziare con le pratiche che si possono adottare *da subito*: principalmente quelle tecniche relative alla codifica. Successivamente, diffuse con successo queste pratiche si possono proporre *brown-bag sessions* e iniziare a parlare del ritmo di sviluppo (iterazioni).

`THE END`