Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Informatica

Essence: un metamodello per processi di sviluppo

Relatore:
Chiar.mo Prof.
Paolo Ciancarini
Correlatore:
Chiar.mo Prof.
Marcello Missiroli

Presentata da:
Paolo Filippini

Sessione
2020/2021

Indice

Elenco delle figure

Introduzione

In questo testo si spigheranno le carte Essence, cosa sono, la loro storia, il metodo e la comunità che ne è dietro. Si cerca di dare un idea del potenziale di Essence, perché è fondamentale in un futuro. Verrà spiegato come utilizzare le carte, i giochi e le carte insieme alla metodologia Scrum. Racconterò la mia esperienza nel progetto di ingegneria del software 2020/2021 ed il ruolo delle carte. Infine mostro e spiego le mie carte Essence, sul software GitLab mostrando come avrebbero potuto aiutare il mio team nella retrospettiva.

La tesi è formata da quattro capitolo. Così suddivisi

1. Primo capitolo

   Il primo capitolo si apre, con la storia del metodo Essence, fondamentale per capire da chi è nato e in quale periodo storico. Successivamente viene presentato il kernel. Dopo questa breve introduzione si entra nel vivo del capitolo, spiegando e facendo vedere le carte vere e proprie, si farà luce su alcuni dettagli, come colore o significato dei simboli. Si passerà alla visione dei sette alfa, rendendo così il kernel tangibile, argomento molto importante sarà ripreso più volte durante l'elaborato. Dove aver visto gli alfa vengono illustrati gli altri componenti del kernel, come competenza, prodotti del lavoro, attività e spazi di attività, pattern ed infine risorse. Il primo capitolo si conclude con un riassunto dei componenti e di come si legano fra di loro.

2. Secondo capitolo

   Il secondo capitolo verte totalmente sull'uso delle carte. Spiego i sette giochi, come funzionano e le regole, quando andrebbero usati, eventuali pro e contro. Viene poi introdotto la metodologia Scrum e come le carte aiutino a migliorare questo metodo, il paragrafo non è esaustivo, è un buon punto di partenza, sopratutto per chi non conosce Scrum. Nell'ultimo paragrafo si vede Essence come teoria dell'ingegneria del software e del perché l'ingegneria del software ha bisogno di una teoria.

3. Terzo capitolo

   In questo capitolo racconto la mia esperienza con le carte Essence nel progetto di ing. del software 2020/2021. Dopo un introduzione al progetto e alla sua struttura, illustro come abbiamo usato le carte e faccio un confronto con un altro team, valutando pro e contro di entrambi i metodi di utilizzo

4. Quarto capitolo

   Questo capitolo verte nel'illustrare le mie carte GitLab, per fare ciò prima di mostrarle spiego alcuni aspetti di GitLab legati al'uso in team, utili per comprendere al meglio le carte. Mostro e motivo le carte, ed infine mostro l'impatto delle carte, in un team che conosce poco GitLab.

Capitolo 1

Il modello Essence

In questo capitolo presenteremo il modello Essence, la sua storia, come e perché è nato. Vedremo il kernel di Essence, e come viene reso praticabile e tangibile tramite le carte. Vedremo i vari tipi di carte ed il loro significato. Per scrivere questo capitolo ho letto il libro *The Essentials of Modern Software Engineering* [3].

1.1 Come tutto è iniziato

La prima domanda che ci facciamo è la più semplice: cosa sono le carte Essence?

Le carte servono per aiutare i team di sviluppo nei loro progetti. Queste carte nascono nel 2009, da diversi contributi della comunità dell'ingegneria del software e di alcuni studiosi che si sono riuniti, su iniziativa di Ivar Jacobson, per discutere il futuro della disciplina [4]. Ricordiamo che Jacobson è stato negli anni '90 uno degli autori di UML.

L'iniziativa SEMAT (*Software Engineering And Theory*) è iniziata oltre che con Jacobson con altri due leader che l'hanno fondata: Bertrand Mayer e Richard Soley.

L'appello del comitato SEMAT nel 2009 recitava quanto segue:

"L'ingegneria del software è gravemente ostacolata oggi da pratiche immature. I problemi specifici includono:

- La prevalenza delle mode, che è più tipica dell'industria della moda che di una disciplina ingegneristica.
- La mancanza di una base teorica solida e ampiamente accettata.
- L'enorme numero di metodi e varianti di metodi, con differenze poco chiare e ingrandite artificialmente.

- La mancanza di una valutazione e convalida sperimentale credibile.

- La separazione tra la pratica industriale e la ricerca accademica

Sosteniamo un processo per rifondare l'Ingegneria del Software basato su una solida teoria, principi provati e buone pratiche che:

- Includano un nucleo di elementi ampiamente concordati ed estendibili per vari usi

- Siano capaci di affrontare i problemi tecnologici e delle persone

- Siano sostenuti dall'industria, dal mondo accademico, dai ricercatori e dagli utenti

- Possano ammettere estensioni di fronte al cambiamento dei requisiti e della tecnologia."

Nel 2011 il comitato SEMAT, dopo aver lavorato due anni e aver raggiunto parte di una proposta per un terreno comune, ha deciso di proporre come uno standard formale ad un organismo di standard accreditato. La scelta cadde su OMG. Tuttavia, ci sono voluti altri tre anni per farlo passare attraverso il processo di standardizzazione. Infine, il linguaggio sottostante e il kernel dell'ingegneria del software [4] sono stati accettati dall'OMG nel giugno 2014 come standard e gli è stato dato il nome di *Essence – Kernel and Language for Software Engineering Methods*, oggi alla versione 1.2 [5].

1.2 Il kernel

Come prima cosa spieghiamo la differenza fra pratica e metodo.

Solitamente un metodo è un insieme di pratiche (esempi: Extreme Programming oppure Scrum), mentre una pratica (composita o meno) è solo un aspetto di ciò di cui il team ha bisogno per affrontare tutte le "cose" che devono essere fatte quando si sviluppa software (esempi di pratica: Pair Programming oppure Incontri giornalieri).

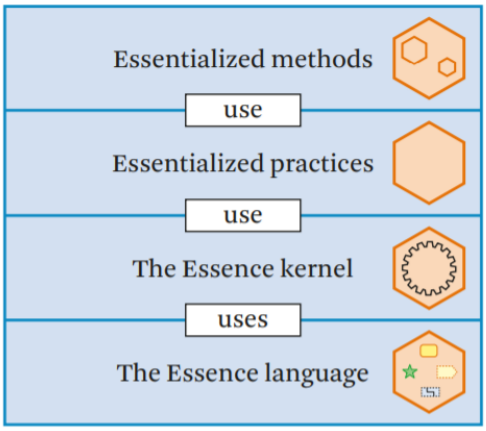

Figura 1.1: L'architettura del metodo Essence.

La Fig.1.1 mostra la notazione usata nel linguaggio Essence: per le pratiche è l'esagono, e per i metodi è l'esagono che racchiude due esagoni minori (sta a significare che i metodi sono insiemi di pratiche). Le pratiche come anche i metodi sono "essenzializzati", vale a dire che sono descritti utilizzando Essence cioè il suo Kernel e il suo linguaggio.

"Essenzializzare" non significa solo che il metodo o pratica è descritto usando Essence ma che focalizza la descrizione del metodo/pratica su ciò che è essenziale, senza cambiare l'intento della metodo/pratica [3].

Il kernel per come descritto finora è qualcosa di astratto, ed è qui che entrano in gioco le carte Essence, le carte servono per rendere il Kernel tangibile.

1.3 Introduzione alle carte

Essence si basa su concetti "primitivi" chwe si chiamano "alpha".

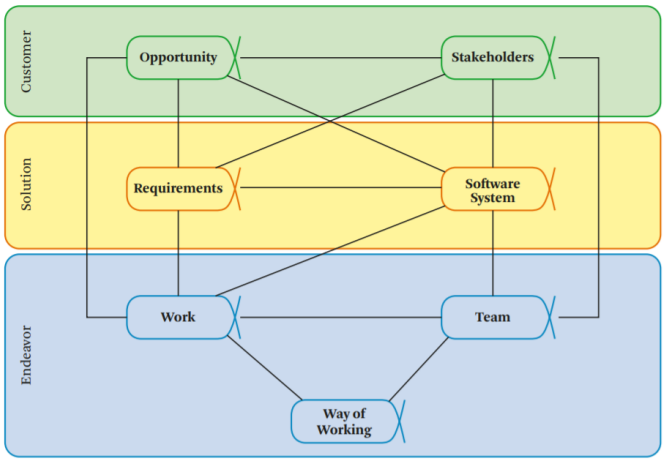

Figura 1.2: Gli ambiti coinvolti in tutto il processo di sviluppo.

Nella Fig.1.2 vediamo i sette **alfa**. Si chiamano alfa perché il simbolo che li rappresenta è simile alla lettera greca (α). Questi sette sono quelli canonici, ma poi ogni team può aggiungere degli altri alfa (con le relative carte) se lo ritiene opportuno.

La prima cosa che si nota è la suddivisione in colori: verde, giallo e azzurro. Questi colori come indicato nella figura rappresentano i tre principali ambiti di un progetto e sono:

- Clienti (Verde), con esigenze da soddisfare
  - Qualcuno ha un problema o un'opportunità da affrontare.
  - Ci sono parti interessate che usano e/o beneficiano della soluzione prodotta, e alcune di queste finanzieranno il progetto.
- Soluzione (Giallo), da consegnare
  - Ci sono alcuni requisiti da soddisfare.
  - Verrà sviluppato un sistema software in base all'esigenza
- Progetto (Blu), da intraprendere
  - Il lavoro deve essere iniziato.
  - Si deve formare un team di persone competenti, che seguano un modo di lavorare condiviso e appropriato.

Ore passiamo ad esaminare in dettaglio ciascun alfa.

- Opportunità: Un'opportunità è una possibilità di fare qualcosa, inclusa la correzione di un problema esistente. Indipendentemente da quale sia l'opportunità, essa può avere successo o fallire. Può fornire un valore reale o può essere qualcosa che nessuno vuole. Quando si lavora su un'opportunità è importante valutare continuamente la fattibilità dell'opportunità man mano che viene implementata.

  Quando un opportunità è concepita per la prima volta, è necessaria una certa diligenza per determinare se si rivolge veramente a un bisogno reale o a una nuova idea reale per la quale i clienti sono disposti a pagare. È probabile che siano disponibili diverse opzioni di soluzione per affrontare l'opportunità, e che si debbano fare alcune scelte difficili.

- Stakeholders: Gli stakeholder si dividono in due categorie: quelli che hanno qualche interesse o preoccupazione nel sistema che si sta sviluppando, sono conosciuti come stakeholder esterni (finanziatori); quelli interessati allo progetto stesso sono chiamati stakeholder interni (team di sviluppo).

  Una delle più grandi sfide in uno progetto di sviluppo è mettere d'accordo tutte le parti interessate. La cosa peggiore che potrebbe accadere è che quando tutto è stato detto e fatto, qualcuno dice: "Questo non è quello che vogliamo". Questo succede troppo spesso.

  Quindi, è molto importante all'**inizio** dell'impresa:

  - capire quali sono le parti interessate e quali sono le loro preoccupazioni;
  - garantire che le parti interessate siano adeguatamente rappresentate e coinvolte nel processo;
  - assicurarsi che siano soddisfatte della soluzione in evoluzione.

- Requisiti I requisiti forniscono il punto di vista delle parti interessate su ciò che si aspettano che il sistema software fornisca. Indicano ciò che il sistema software deve fare, ma non esprimono esplicitamente come deve essere fatto. Identificano il valore del sistema rispetto all'opportunità e indicano come l'opportunità sarà perseguita attraverso la creazione del sistema.

  Tra le più grandi sfide che i team di software devono affrontare ci sono i requisiti mutevoli. I requisiti possono cambiare per un infinità di motivazioni, dovuti anche agli stakeholders che avranno opinioni diverse e in conflitto fra di loro, quindi il software system si evolverà insieme ai requisiti.

Il modo in cui un team lavora con i requisiti è assolutamente cruciale. I principi che dovrebbe seguire sono i seguenti:

  - assicurare che i requisiti siano continuamente ancorati all'opportunità;
  - organizzare i requisiti in modo da facilitare la comprensione e risolvere i requisiti conflittuali;
  - assicurare che i requisiti siano testabili, cioè che si possa verificare che il sistema software soddisfi effettivamente i requisiti senza ambiguità;
  - usare i requisiti per guidare lo sviluppo del *software system*. Il codice deve essere ben strutturato e facile da riferire ai requisiti. Il progresso è misurato in termini di quanti dei requisiti sono stati completati.

- Software System Il risultato principale di un progetto di sviluppo software è naturalmente il *software system* stesso. Il sistema software può presentarsi in molte forme diverse: per esempio un sito web o applicazione mobile, e potrebbe girare su un sistema distribuito o su una singola macchina. Qualunque sia il caso, ci sono tre importanti caratteristiche dei sistemi software necessarie prima che possano essere di valore per gli utenti e le parti interessate: *funzionalità, qualità ed estensibilità*.

Funzionalità : I sistemi software sono progettati e costruiti per facilitare la vita degli esseri umani. Ognuno di essi deve offrire alcune funzioni, che sono derivate dai requisiti del sistema software.

Qualità : È un elemento fondamentale di cui la necessità non va discussa. Naturalmente, il grado di qualità necessario dipende dal contesto stesso. Anche questo può essere derivato dai requisiti del sistema software.

Estensibilità : L'ingegneria del software ha l'obiettivo di facilitare modifiche ed evoluzione del sistema software, da una versione all'altra, dandogli sempre più funzionalità per servire i suoi utenti. Non basta del semplice patching altrimenti, quando il sistema software cresce di dimensione, sarà più difficile aggiungere nuove funzionalità. Di conseguenza, i team spesso organizzano il sistema software in parti interconnesse note come componenti. Ogni componente realizza una parte dei requisiti e ha uno scopo e un'interfaccia ben definiti, il codice deve essere ben strutturato e facile da ricollegare ai requisiti. Come ultima cosa l'evoluzione comporta la transizione del sistema software attraverso diversi ambienti, ad esempio dalla macchina dello sviluppatore a

quella del cliente. Dunque il lavoro di uno sviluppatore non è finito finché il sistema non funziona bene nell'ambiente di produzione. Quindi la frase: "*Ma funziona sulla mia macchina*" non è una scusa accettabile.

Per concludere un sistema software di qualità deve:

* avere un progetto che sia una soluzione al problema e concordato
* essere utilizzabile, aggiungendo valore alle parti interessate
* avere un supporto operativo sul posto

- Team

  Un buon lavoro di squadra è essenziale per ottenere alte prestazioni. Crea una sinergia, dove l'effetto combinato della squadra è molto più grande della somma degli sforzi individuali.

  Raggiungere uno stato di alta prestazione non è facile, ma è il risultato di un tentativo deliberato di avere successo.

  Per ottenere questo alto livello di prestazioni, i membri del team dovrebbero riflettere sul modo in cui lavorano insieme e su come si concentrano sull'obiettivo di team.

  Il team ha bisogno di:

  – essere formato da un numero sufficiente di persone per iniziare il lavoro
  – essere composto da personale che possiede le giuste competenze/abilità
  – lavorare insieme in modo collaborativo
  – sapersi adattare continuamente all'ambiente che cambia

- Lavoro I membri del team devono essere in grado di preparare, coordinare, seguire e completare (fermare) il loro lavoro. Il successo in questo ha un effetto profondo sul rispetto degli impegni e sulla fornitura del valore (software system) agli stakeholder. Quindi, i membri del team devono capire come eseguire il loro lavoro e riconoscere se il lavoro sta procedendo in modo soddisfacente.

  Per fare ciò bisogna:

  – prepararsi
  – comunicando il lavoro da fare
  – assicurare che il progresso e il rischio siano sotto controllo
  – sapere quando chiudere il lavoro

- Modo di lavorare

Un team può svolgere il suo lavoro in modi diversi e questo può portare a risultati diversi. Può essere eseguito *ad hoc*, il che significa che si decide come lavorare mentre si fa il lavoro. Quando si segue un modo di lavorare ad hoc, il risultato può essere di alta qualità o meno. Questo dipende da molti fattori: tra questi, l'abilità delle persone coinvolte e il numero di persone coinvolte nel processo. Se troppe persone decidono come deve essere fatto il lavoro, probabilmente non verrà fuori un buon metodo di lavoro. Ricordiamo il detto: "troppi cuochi rovinano il brodo". Quindi il modo di lavorare deve:

- includere una base di pratiche e strumenti chiave
- essere usato da tutti i membri del team
- essere migliorato dal team quando necessario

Qui si conclude la spiegazione dei sette alfa, lunga ma necessaria perché li useremo molto anche in seguito.

1.4 Il linguaggio Essence

Essence fornisce un linguaggio preciso e utilizzabile per descrivere le migliori pratiche di ingegneria del software. Ad esempio, nella Fig.1.3 vediamo il *pair programming* (programmazione a coppie) descritto mediante il linguaggio Essence.

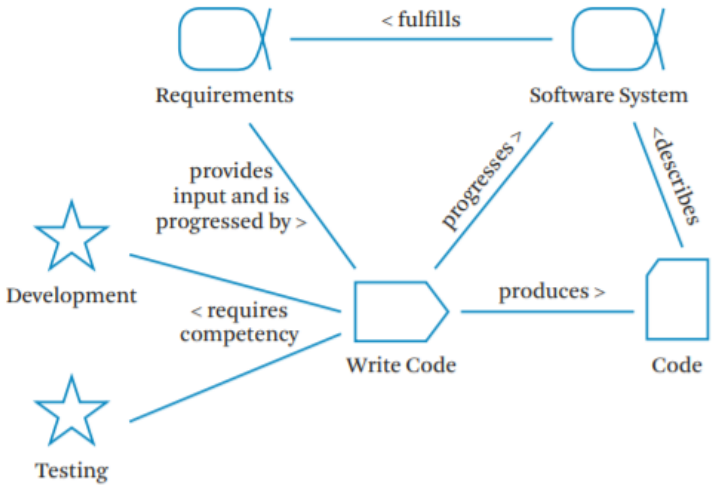

Figura 1.3: Pratica di programmazione a coppie descritta usando il linguaggio Essence

Ora diamo una piccola descrizione a parole:

Lo scopo di questa pratica è produrre codice di qualità migliore di quello che si produrrebbe singolarmente. In questo caso, assumiamo che il concetto di qualità del codice sia comprensibile ai diversi membri del team.

Due persone (es. due studenti) lavorano in coppia per trasformare i requisiti in un sistema software scrivendo codice insieme.

Scrivere codice fa parte dell'implementazione del sistema.

Come possiamo notare dall'immagine ritroviamo due alfa già visti, i requisiti e il sistema software, ma vediamo anche simboli mai visti prima come una stellina e altri.

Ora procediamo con la spiegazione e iniziamo con l'immagine riassuntiva in Fig.1.4:

Element Type	Syntax	Meaning of Element Type
Alpha		An essential element of the development endeavor that is relevant to an assessment of the progress and health of the endeavor.
Work Product		A tangible thing that practitioners produce when conducting software engineering activities.
Activity		A thing that practitioners do.
Competency		An ability, capability, attainment, knowledge, or skill necessary to do a certain kind of work.

Figura 1.4: Elementi del linguaggio Essence

Ora andiamo a vedere nel dettaglio cosa significano questi nomi e questi simboli.

- alfa Gli alfa sono gli elementi più importanti a cui prestare attenzione per avere successo in un'impresa di sviluppo software. Anche se un'alfa è comunemente compresa o evidenziata da uno o più prodotti di lavoro associati, non è tangibile di per sé, quindi ci deve essere almeno qualche conoscenza tacita legata ad ogni alfa. Per questo usiamo gli *alfa state*.

  Gli alfa hanno degli stati che descrivono la progressione attraverso un ciclo di vita.

  Nella Fig.1.5 vediamo la carta di un alfa, ha un nome e il simbolo che sta indicare che è un alfa, una descrizione e i suoi *alfa states*.

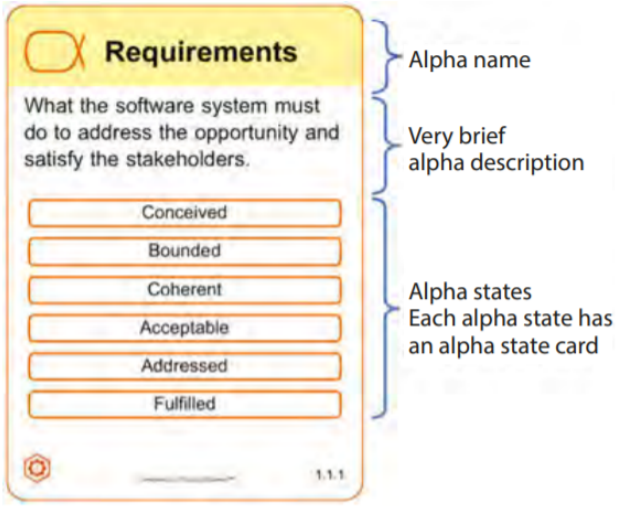

Figura 1.5: Carta alfa dei requisiti.

Come possiamo notare la carta dei requisiti ha sei stati. Un alfa state è formato dall'alfa a cui fa riferimento il nome proprio e i campi di controllo riguardanti quello stato. I campi servono al team per capire se sono in quello stato o meno, e se sì a che punto sono. Gli alfa state dei requisiti sono mostrati in Fig.1.6.

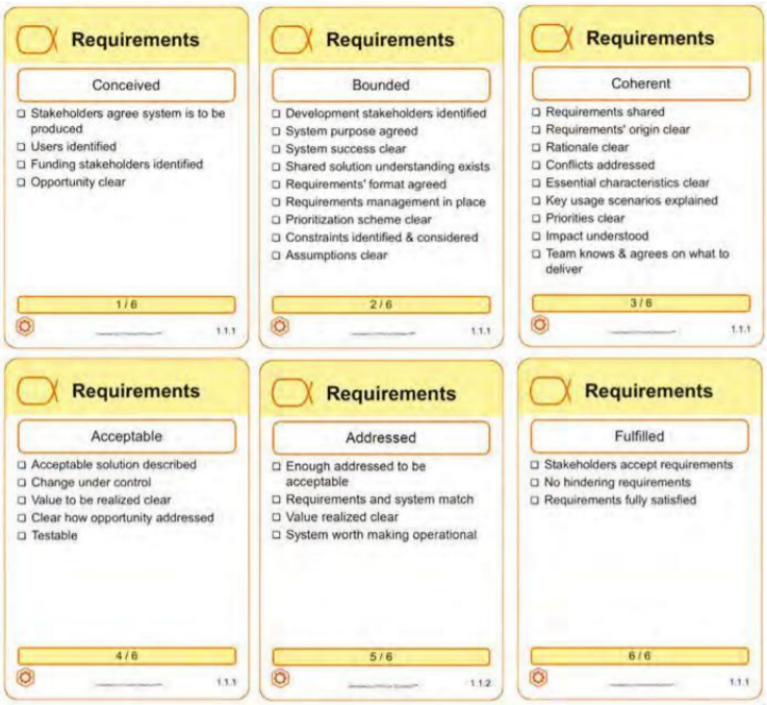

Figura 1.6: Carte dei diversi stati in cui possono trovarsi i requisiti.

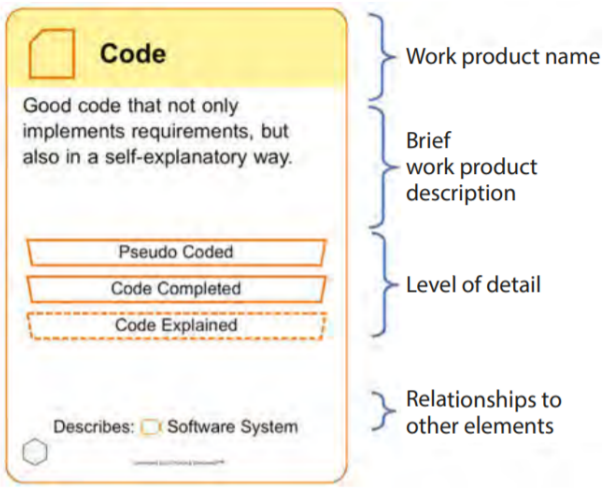

Figura 1.7: Carta di un prodotto di lavoro: Code

Notare l'articolazione in punti selezionabili all'interno di ciascuna carta-stato, che aiuta a determinare se un requisito si trova nello stato stesso.

- Prodotti di lavoro

  I prodotti di lavoro (o artefatti) sono elementi tangibili come documenti e rapporti, e possono fornire elementi per verificare il raggiungimento degli alfa states. Il codice è l'esempio di un prodotto di lavoro per la pratica di programmazione Pair programming - vedi Fig. 1.3.

  Si noti che i prodotti di lavoro non appartengono al terreno comune rappresentato dallo standard Essence. Essi dipendono da come si vuole lavorare (cioè da quali pratiche vengono usate). Quindi, Essence non specifica esattamente quali prodotti di lavoro devono essere sviluppati, ma specifica cosa è un prodotto di lavoro, come si rappresenta e cosa fare con essi.

  I prodotti di lavoro hanno diversi livelli di dettaglio che dipendono dal progetto e dal team e da altri fattori, come anche da fattori legali (esempio: cercare di conseguire una certificazione ISO).

- Competenze

  Una competenza comprende le abilità, le capacità, i risultati, le conoscenze e le abilità necessarie per fare un certo tipo di lavoro. Si presume che per ogni membro del team possa essere determinato un livello di competenza per ogni singola competenza.

Un livello di competenza definisce un livello di quanto competente o capace sia un membro del team rispetto a una determinata abilità.

I team dovrebbero essere incoraggiati a condurre un'autovalutazione delle loro competenze e confrontare i risultati con le competenze di cui credono di aver bisogno per portare a termine il loro impegno specifico. Questo utile esercizio può aiutare i team di sviluppo software a determinare obiettivamente qualsiasi lacuna di competenza che possono avere, che possono segnalare alla direzione per la risoluzione prima che si verifichino problemi seri che potrebbero danneggiare le prestazioni del team.

La Fig.1.8 mostra una carta competenza.

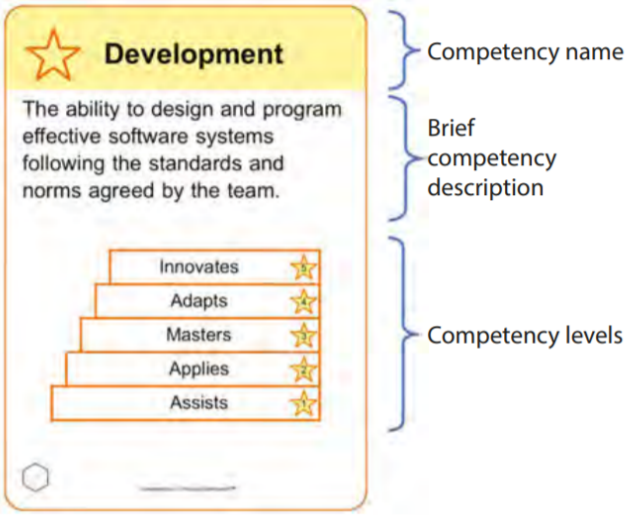

Figura 1.8: Carta competenza: Development

Ogni team ha bisogno di competenze diverse. Essence definisce solo sei competenze perché assume che queste siano trasversali per ogni team. Vediamo in Fig.1.9 le tre principali categorie classificate mediante i colori.

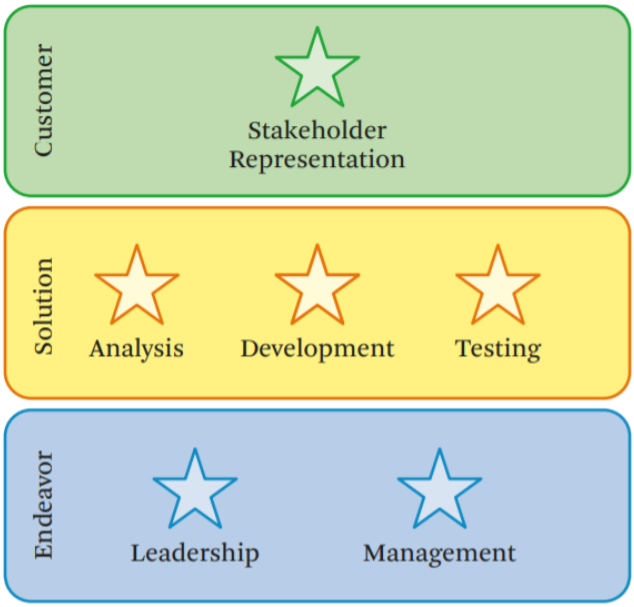

Figura 1.9: Competenze. Sono mostrate sei competenze raggruppate nelle diverse aree colorate Customer, Solution ed Endeavor. Ad esempio la soluzione richiede competenze di Analisi, Sviluppo e Testing

- Attività

Le attività (*activity*) sono azioni che i professionisti fanno, come tenere una riunione, analizzare un requisito, scrivere codice, testare o fare una revisione tra pari. Le attività sono specifiche e non standard - non fanno parte di Essence. Un'attività è sempre legata ad una specifica pratica e non può "fluttuare" tra le pratiche. Se trovi un'attività che ha bisogno di essere riutilizzata da molte pratiche, allora potresti voler creare una pratica separata che includa questa attività. L'alternativa è che tu decida di non riutilizzarla, ma ogni pratica che potenzialmente potrebbe riutilizzarla dovrà mantenere la propria copia di quell'attività. Le modifiche ad una copia non avranno impatto sulle altre copie; esse cambieranno indipendentemente l'una dall'altra, il che significa nessun riutilizzo. In Fig.1.10 vediamo la carta di un'activity: notare che ha un input, e che fornisce un output.

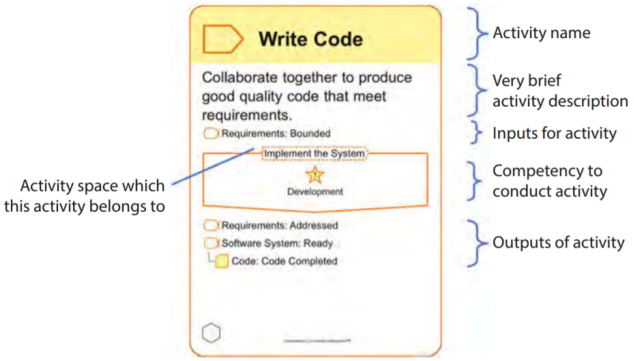

Figura 1.10: Carta attività della codifica (*Write code*)

- Spazi di attività

  Gli spazi di attività sono segnaposti generici (cioè indipendenti dal metodo) per attività specifiche che saranno aggiunte in seguito, sopra il kernel. Poiché gli spazi di attività sono generici, possono essere standardizzati e sono quindi parte dello standard Essence. In Fig.1.11 possiamo vedere gli spazi di attività dello standard Essence.

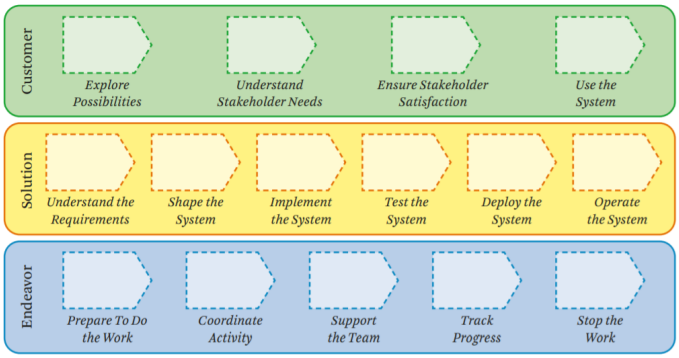

Figura 1.11: Spazi di attività (dallo standard Essence)

Riguardo a questa immagine la sequenza indica l'ordine in cui le cose sono devono essere dichiarate finite e non necessariamente l'ordine in cui sono iniziate. Per esempio, si può iniziare a modellare il sistema prima di aver finito di capire i requisiti, ma non si può essere sicuri di aver finito di modellare il sistema finché non si è finito di capire i requisiti.

Per fare un esempio, l'attività *Write Code* verrebbe assegnata allo spazio attività *Implement the System*.

Oltre ad essere segnaposto per attività specifiche, gli spazi di attività rappresentano gli elementi essenziali che devono essere fatti per sviluppare software.

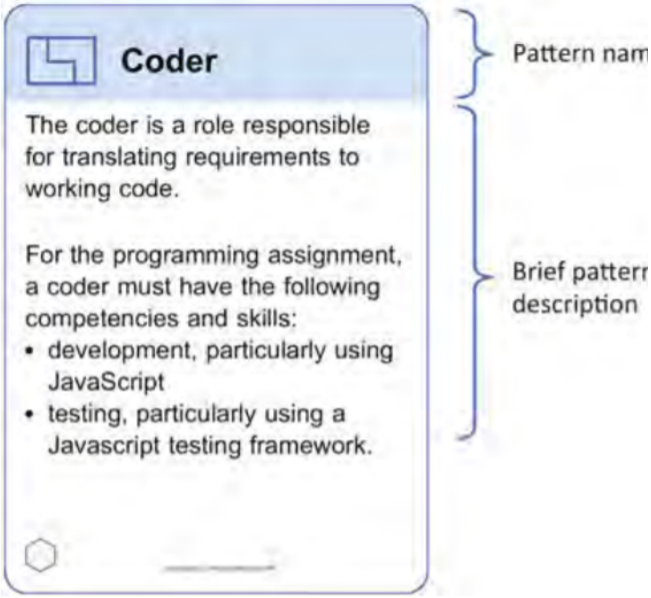

Figura 1.12: Pattern con competenze: il coder deve saper sviluppare e testare

- Pattern (Modello)

  I pattern sono soluzioni generiche a problemi tipici e ricorrenti. Un tipico problema da affrontare potrebbe essere come assegnare le responsabilità associate agli incarichi di lavoro ai membri del team. I pattern sono elementi opzionali (non è un elemento obbligatorio nella definizione di una pratica) che possono essere associati a qualsiasi altro elemento del linguaggio.

  Molti tipi di lavoro richiedono più di una competenza. Un modo comune per assicurarsi che un individuo assegnato a questo lavoro abbia queste competenze è quello di definire un tipo speciale di pattern, chiamato *ruolo*. Un ruolo designa non solo un insieme di responsabilità specifiche, ma anche le competenze richieste per svolgerle. Un ruolo può anche specificare un livello minimo di competenza necessaria per fare il lavoro in modo efficace.

  La Fig.1.12 mostra un esempio di una carta pattern, in cui sono richieste certe competenze.

  Un altro modo per usare i pattern è quello di essere usato per definire un checkpoint. Un checkpoint è un insieme di criteri da raggiungere in un momento specifico in un'impresa di sviluppo. Sono usati dalle organizzazioni come punti in cui si fermano e pensano se ha senso procedere o meno, a seconda che ci sia o meno fiducia in ciò che è stato fatto. Una decisione potrebbe anche essere presa

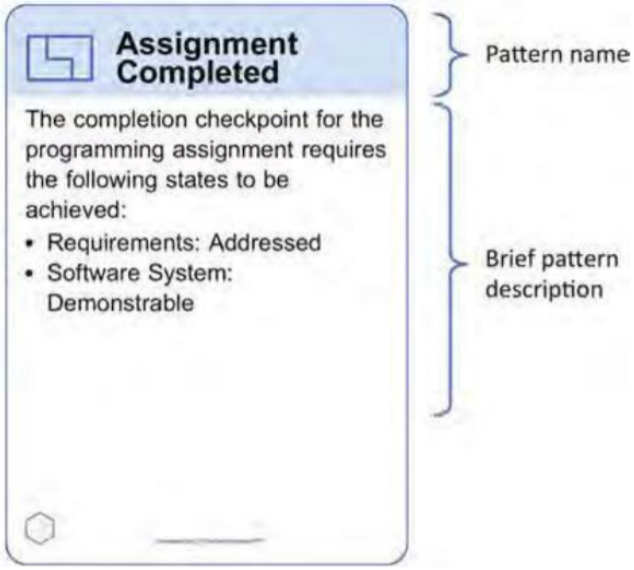

Figura 1.13: Pattern con checkpoint

per andare avanti ma in una direzione diversa a causa di un problema o di un rischio noto.

La Fig.1.13 mostra una carta pattern che usa i checkpoint (invece che le competenze)

- Risorse

Una risorsa è una fonte di informazioni o contenuti, come per esempio un sito web, che è al di fuori del modello Essence e a cui si fa riferimento, per esempio tramite un URL.

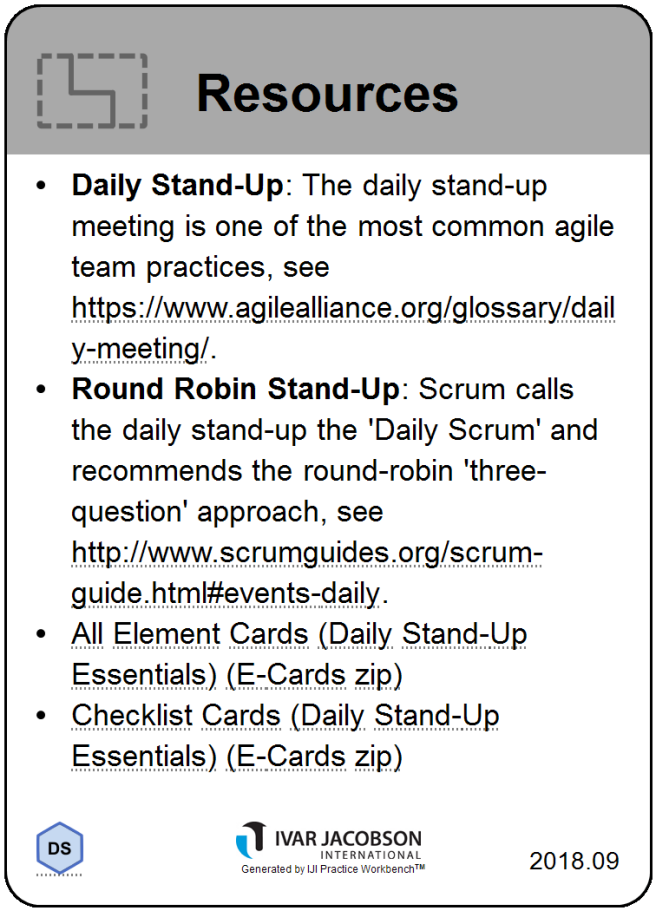

Figura 1.14: Esempio di una carta risorsa

1.5 Essence visto dall'alto

Adesso che abbiamo visto tutti gli elementi del linguaggio Essence, mostriamo in Fig.1.15 un grafico che mostra la relazione fra tutti gli elementi.

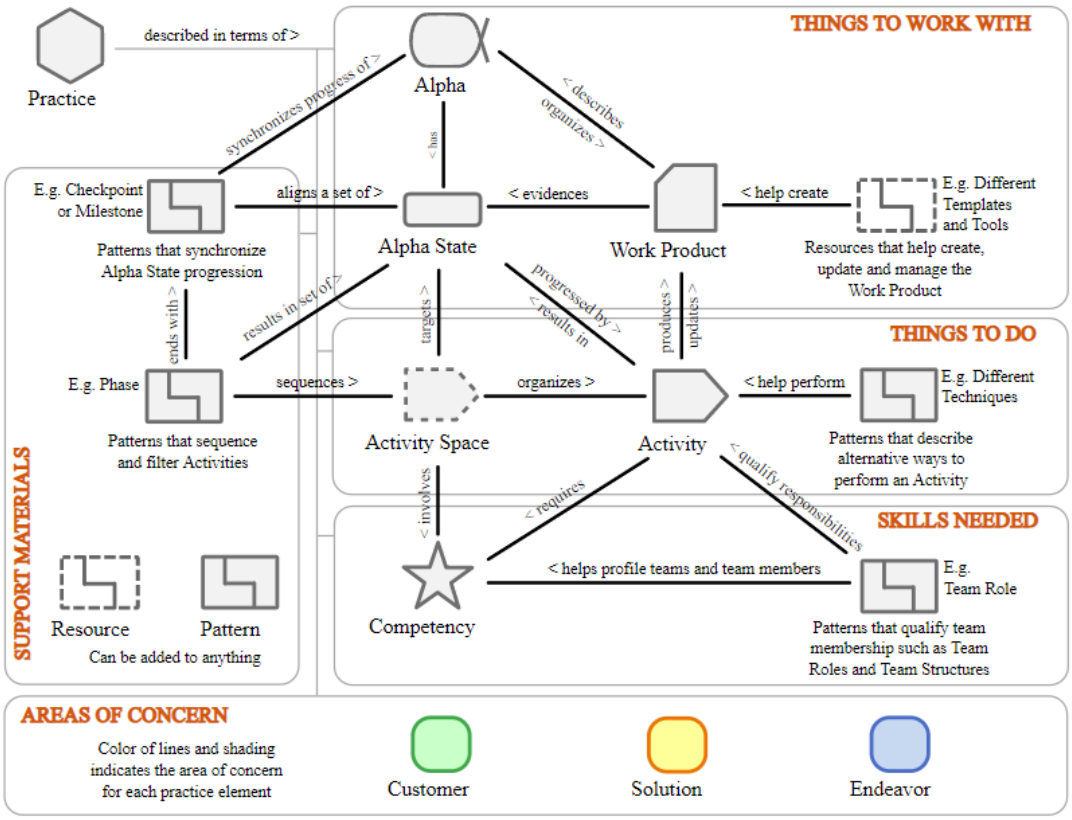

Figura 1.15: Schema del linguaggio Essence [1]

Notiamo in alto a sinistra un esagono, simbolo della pratica, che ci dice che una pratica è descritta da tutti quegli insiemi di elementi (un Metodo è un insieme di pratiche).

Notiamo che ci sono quattro scritte in arancione che sono le quattro macro categorie, i nomi sono auto esplicativi ma vediamoli in breve:

- Cose con cui lavorare (*things to work with*) comprendono gli alfa e i loro stati e i prodotti di lavoro e le risorse.

  In generale si parla di cose con cui si lavora direttamente, in concreto. Ad esempio il codice scritto (prodotto di lavoro)

- Cose da fare (*things to do*) comprendono le attività, gli spazi di attività, e i pattern. Vedremo che i pattern sono compresi in più categorie.

  In generale si parla di cose che vanno fatte (non di come vanno fatte): sono più astratte, vero è che le attività si eseguono ma l'attività in se è qualcosa di immateriale.

- Competenze necessarie (*skill needed*) Naturalmente ci sono le competenze, e vediamo ancora i pattern ma questa volta usati più come ruoli e struttura del team.

- Materiale di supporto (*support material*) Queste sono risorse, e anche qui vediamo i pattern usati come supporto, ad esempio per sincronizzare gli stati degli alfa dopo dei progressi, o come aiuto nelle attività.

- Aree di interesse Di questo ne abbiamo già parlato a inizio capitolo. Come possiamo vedere è legato solo alla pratica, sta indicare che tutte le carte Essence sono obbligatoriamente in un area di interesse.

Ora iniziamo a leggere le frecce. Gli alfa organizzano i prodotti di lavoro, in base a quale alfa prendiamo all'inizio e quale vogliamo raggiungere decidiamo quale prodotto di lavoro fare. Poi i prodotti di lavoro descrivono gli alfa, che a loro volta hanno degli stati. Gli stati degli alfa sono progrediti dalle attività questo vuol dire che è tramite le attività che mandiamo avanti gli stati degli alfa(e quindi il nostro progetto), le attività producono dei prodotti di lavoro o aggiornano quelli già esistenti, le attività vengono aiutate dai pattern che specificano i modi alternativi con cui può essere eseguita una attività,per eseguire un attività è richiesta una o più competenze. Le attività sono organizzate tramite gli spazi di attività, ogni spazio di attività ha come obbiettivo uno stato di un alfa e coinvolge una o più competenze.

Abbiamo spiegato il rettangolo centrale più grande. Ora diciamo qualcosa anche sul rettangolo a sinistra. Qui ci sono due pattern, usati in due modi diversi, uno per sequenziare e filtrare le attività un altro per sincronizzare i progressi degli stati alfa. Il primo è in relazione con gli spazi di attività perché come detto appunto ne definisce una sequenza. Il secondo è in relazione con gli alfa e con gli stati ala poiché il suo compito è quello di sincronizzarli in seguito a dei progressi fatti.

1.6 Conclusioni

In questo capitolo abbiamo visto la storia di Essence e del motivo della sua nascita. Il kernel che è il cuore della teoria e di come viene reso tangibile e praticabile tramite le carte. Abbiamo spiegato tutte i sette alfa standardizzati da Essence, molti importanti essendo le prime carte standardizzate ed inserite all'interno del metodo. Siamo poi passati alla spiegazione delle cinque tipologie di carte, del loro significato e della differenza che intercorre fra di esse, tutto con esempi di carte reali. Il capitolo si conclude con una

overview generale di Essence e di come le le diverse tipologie di carte interagiscono fra di loro. Nei prossimi capitoli, seguendo la proposta [6], vedremo come abbiamo usato Essence.

Capitolo 2

Essence in pratica

Dopo aver capito cosa sono e che scopo hanno le carte Essence, spieghiamo come usarle durante una retrospettiva Scrum.

Il primo paragrafo spiega dei giochi su come usare gli alfa, oltre alla descrizione del gioco c'è anche scritto quando è consigliabile giocare a tale gioco. Nel secondo paragrafo viene fatto un esempio di come viene usato Essence, insieme a Scrum. L'ultimo paragrafo spiega perché è necessaria una teoria e come Essence stia provando a definirla.

2.1 Come usare gli alfa

I team di sviluppatori usano le carte Essence come strumenti di facilitazione in una varietà di impostazioni e scopi: per esempio, per dialogare e ottenere un consenso sullo stato del lavoro. Le carte sono anche un buon modo per introdurre gli elementi del kernel e della pratica alle persone che sono nuove del modello Essence.

Tutti i membri del team devono capire e concordare lo scopo del progetto e i suoi benefici, così come i problemi di sviluppo, e risolvere eventuali conflitti entro un tempo limitato. Per questo motivo, la maggior parte dei giochi seri utilizzati nell'ingegneria del software sono di natura collaborativa.

I giochi seri possono essere divertenti, ma il loro scopo va oltre l'intrattenimento. Simulando eventi realistici, i giochi seri mirano a raggiungere un obiettivo specifico.

Fare giochi seri con le carte Essence è un modo per aiutare i membri del team ad osservare come ragionano i compagni di squadra, e aiutarli a far progredire il loro ragionamento in modo strutturato e sistematico. Questo perché utilizzando parole naturali usate dalla maggior parte dei team di sviluppo nel nominare gli alfa, gli stati e le liste di controllo (*checklist*), le carte stimolano un team a discutere le questioni relative alla salute e al progresso dei propri sforzi.

Ora vediamo sette giochi seri che possiamo fare con le carte alfa.

2.1.1 Progress Poker

Questo gioco permette di determinare lo stato di un particolare alfa. Una delle domande più importanti che i team devono spesso affrontare è "Abbiamo finito?" riferita ad una particolare parte di lavoro completata, ed alla definizione di *fatto* (Definition of Done).

Questo di solito scatena lunghe discussioni sulla "definizione di fatto". Mentre ci sono diverse definizioni di fatto, la nostra si riferisce al movimento di un stato alfa ad un altro stato.

È vero che gli elementi della lista di controllo non forniscono una definizione precisa. Se lo facessero, probabilmente sarebbe incomprensibile per la maggior parte degli sviluppatori.

Gli elementi non sono precisamente definiti, ma forniscono un indizio di ciò che deve essere fatto. Sono soggetti all'interpretazione dei membri del team, che possono avere opinioni diverse sul loro significato. Sarebbe buona cosa, concordare una definizione di fatto, prima di giocare con gli alfa. Un modo per raggiungere un accordo sullo stato di un alfa è giocare a Progress Poker.

Il gioco è così strutturato. Si gioca un'alfa alla volta. Gli strumenti di questo gioco sono un mazzo di carte Essence (in questo caso un mazzo per ogni giocatore), un tavolo.

Progress Poker è un gioco basato sul consenso. Il suo obiettivo principale è quello di assicurarsi che tutti siano allineati. Per giocare a Progress Poker, si ha bisogno della carta panoramica alfa e delle carte stato alfa per il particolare alfa di cui si sta discutendo, come nell'esempio mostrato in Fig. 2.1.

Figura 2.1: Carte necessarie per giocare a Progress Poker (alfa e suoi stati alfa).

In questo gioco non c'è un vincitore. Il vincitore è il team, e la mano vincente è l'accordo comune del team sullo stato dello sviluppo.

Il Progress Poker può essere giocato da qualsiasi numero di giocatori. Tuttavia, l'esperienza ha indicato che è più efficace in team composti da tre a nove giocatori, come quelli di Scrum.

I giocatori, i membri del team, si riuniscono intorno al tavolo. Per comunicare che l'alfa dei requisiti è in esame, mettono la sua carta alfa al centro del tavolo. Ogni giocatore usa poi il proprio set di carte stato per quell'alfa, e identifica dal proprio set la carta che pensa rappresenti meglio lo stato attuale. Fedeli al nome del gioco, dovrebbero mantenere una faccia da poker, cioè tenere riservata la carta da loro scelta, e non far trapelare niente dalla loro espressione (come a poker). Dopo aver scelto la propria carta, ogni giocatore dovrebbe metterla a faccia in giù sul tavolo e aspettare che tutti i membri della squadra abbiano fatto lo stesso. Così facendo, si assicurano che l'opinione iniziale di ognuno non sia influenzata dall'opinione di qualcun altro. Dopo che ognuno ha scelto una carta stato alfa, tutti i giocatori girano le carte stato scelte a faccia in su nello stesso momento e confrontano i risultati. Come ovvio se hanno girato tutti la stessa carta il turno finisce in quanto c'è già un consenso, in caso contrario si intavola una discussione in cui di solito, quelli con lo stato meno avanzato e quello più avanzato dovrebbero iniziare la discussione.

Uno dei vantaggi del Progress Poker è che tutti i membri del team vengono coinvolti, poiché ogni membro della squadra è costretto a fare una valutazione e a spiegare il

proprio punto di vista quando la sua valutazione differisce da quella dei suoi compagni di squadra. Ogni membro della squadra deve pensare e parlare del perché ha valutato lo stato in quel modo. Questo aiuta il team ad evitare decisioni che non sono razionali, ed evita la situazione in cui solo le decisioni di alcuni membri guidano la squadra.

2.1.2 Chasing the State

Questo gioco serve a determinare lo stato del processo di sviluppo del prodotto software.

Questo gioco inizia disponendo tutte le carte di un'alfa su un tavolo. All'estrema sinistra c'è la carta panoramica Alfa con un'immagine di tutti gli stati dell'alfa stesso. A destra ci sono tutte le carte stato dell'alfa con la prima carta stato a sinistra e l'ultima carta stato a destra come nella figura di seguito:

Figura 2.2: Posizione iniziale per il gioco Chasing the State.

Si parte dal primo alfa (*stakeholders*) e dal primo suo stato alfa e tramite la lista di controllo vediamo se lo stato è soddisfatto in caso affermativo spostiamo la carta a sinistra e passiamo ad esaminare il prossimo stato in caso contrario lasciamo lo stato nella sua posizione, così per ogni alfa.

In questo particolare gioco si presume che tutto vada liscio e che il team possa facilmente accordarsi sugli stati che sono stati raggiunti. Questo non è sempre il caso, quindi se il team non si mette subito d'accordo, può giocare a Progress Poker per il particolare alfa che non è facile da concordare.

Consiglio pratico: quando si gioca è meglio dare ad ogni persona un mazzo per ogni stato degli alfa, in questo modo non bisogna prenderle dal centro del tavolo in caso qualcuno voglia rileggere la carta.

2.1.3 Objective Go

Questo gioco serve per identificare nuovi obiettivi e traguardi di alto livello per il team.

Objective Go è giocato per concordare dove si deve andare dopo, ovvero qual è il prossimo passo. Per sapere dove andare dopo, devi ovviamente sapere dove ti trovi. Questo gioco viene giocato dopo aver valutato gli stati attuali di tutti gli alfa. Quindi, di solito si gioca dopo aver giocato il gioco *Chasing the State*. Il team si pone una domanda del tipo:

"Qual è il prossimo insieme di stati alfa che dovremmo raggiungere?"

Il team può decidere che il suo obiettivo è quello di passare allo stato successivo per tutti e sette gli alfa, o può decidere che il prossimo passo dovrebbe essere quello di concentrarsi solo su uno o alcuni degli alfa per progredire verso i loro stati successivi.

In modo simile al *Chasing the State*, il team esamina ogni alfa ritenuto interessante per progredire alla fase successiva. Per ogni alfa, discutono lo stato successivo che dovrebbe essere raggiunto e quali elementi della lista di controllo per quello stato non sono ancora stati raggiunti. Una volta che hanno concordato dove vogliono andare dopo, discutono anche quali compiti devono fare per arrivarci. Per ogni alfa che la squadra ha concordato di progredire verso uno stato successivo, la carta dello stato corrispondente viene spostata al centro del tavolo,

Figura 2.3: Il prossimo passo è rappresentato dalle carte al centro del tavolo.

2.1.4 Checkpoint Construction

Questo gioco serve per definire punti di controllo indipendenti dalla pratica con liste di controllo indipendenti dalla pratica generate automaticamente.

Di solito, le organizzazioni hanno definito dei cicli di vita che consistono in fasi separate da punti di controllo. I punti di controllo sono intenzionalmente indipendenti dalle pratiche che un team usa, perché uno dei loro scopi principali è quello di valutare il progetto da diversi punti di vista come il valore, il finanziamento e la prontezza.

In questo senso, i check-point possono essere visti come punti critici nel ciclo di vita di un progetto dove la definizione di "fatto" per le fasi deve essere specificata. Ad ogni checkpoint, viene presa una decisione se procedere o meno alla fase successiva.

Poiché un progetto può avere molti team che lavorano in parallelo, per sincronizzarsi tra i team, di solito tutti devono avere gli stessi punti di controllo. Così, i punti di controllo sono normalmente specificati dagli stakeholder interni e non dai team che partecipano al progetto.

Checkpoint Construction è giocato per ottenere il consenso sui checkpoint nel ciclo di vita di un progetto. Il numero consigliato di giocatori è fra i tre e i dieci. Ci sono due ruoli:

- Facilitatore: conduce il gioco e le discussioni che ne derivano(uno solo)

- Collaboratore: partecipa al gioco con il facilitatore

Ci sono due turni: nel primo turno, ogni membro del team considera ciascuno dei sette alfa e decide quali devono essere considerati come parte del checkpoint, ognuno annota le proprie scelte. Poi il facilitatore per ogni carta panoramica alfa chiede al team se quell'alfa deve essere considerata nel checkpoint. Ogni giocatore risponde a questa domanda usando un pollice in su o pollice in giù. Così, in questo round il team vota su quali alfa dovrebbero essere considerati per il checkpoint.

La Fig.2.4 mostra un checkpoint di primo turno.

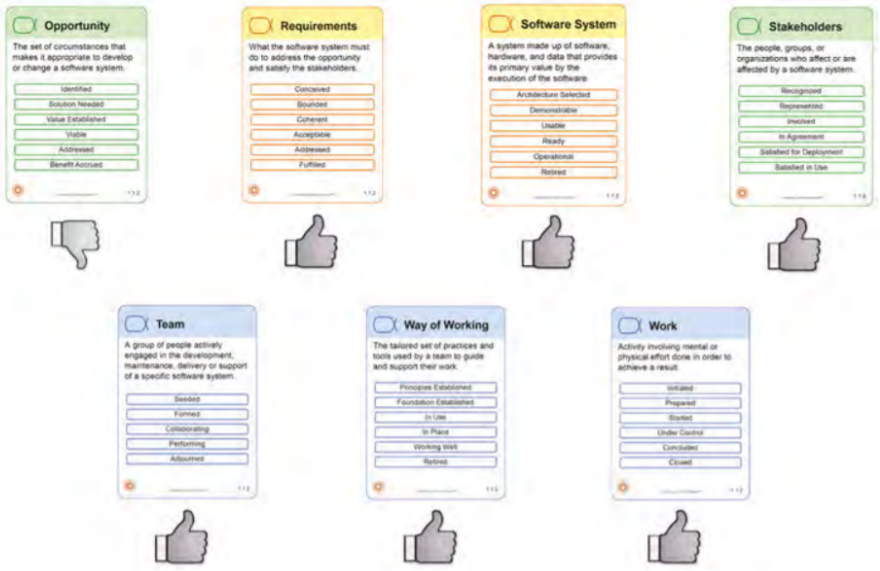

Figura 2.4: Esempio di un primo turno di gioco

Ora il secondo turno, il facilitatore distribuisce tutte le carte stato alfa orizzontalmente sul tavolo per tutti gli alfa selezionati che devono essere considerati per il checkpoint. Ogni giocatore considera l'insieme di stati per ogni alfa e, senza informare gli altri giocatori, identifica lo stato in cui crede che l'alfa debba trovarsi per passare il checkpoint. Quando tutti sono pronti, ognuno dei giocatori alza simultaneamente una mano con il numero di dita che indica lo stato in cui crede che l'alfa debba trovarsi per passare il checkpoint. Il pugno chiuso è usato per indicare il sesto stato. Se tutti i giocatori hanno selezionato lo stesso stato, c'è consenso. In caso contrario, i giocatori con lo stato meno avanzato e quello più avanzato spiegano il loro ragionamento. Dopo la discussione, i giocatori alzano di nuovo simultaneamente le mani, indicando gli stati che hanno selezionato. Questo passo viene continuato fino a quando non si raggiunge il consenso.

Una volta che lo stato è stato concordato, il facilitatore guida il gruppo attraverso una discussione sulle potenziali voci aggiuntive della lista di controllo da aggiungere per questo checkpoint. In questo modo, gli elementi generici della lista di controllo sulle carte possono essere adattati al contesto del progetto specifico

2.1.5 Lifecycle Layout

Questo gioco serve per visualizzare il ciclo di sviluppo del software e formare un punto di partenza per la pianificazione del team.

Le organizzazioni spesso richiedono ai team di usare uno sviluppo software standard o un ciclo di vita standard per lo sviluppo del software o la gestione dei prodotti. Spesso gli standard sono definiti in un modo che limitano eccessivamente la libertà dei team di sviluppo e si concentrano sui prodotti di lavoro specifici da produrre piuttosto che sui risultati o lo stato da raggiungere. Usando le schede alfa questo può essere fatto in modo leggero, focalizzato sul risultato che è indipendente dalla pratica che il team sceglie per raggiungere gli obiettivi.

Questo è un gioco composito che coinvolge diversi round di Checkpoint Construction per definire una serie di checkpoint che rappresentano i punti di revisione nel ciclo di vita. In questo caso però tutti gli alfa devono essere considerati. I giocatori assumono uno dei due ruoli quando giocano a questo gioco:

- Facilitatore (uno solo): conduce il gioco e le discussioni che ne derivano;
- Collaboratore: gioca il gioco sotto la guida del facilitatore.

Il gioco è impostato dal facilitatore e giocato in un certo numero di turni.

Impostazione: Il facilitatore dispone le carte dello Stato alfa sul tavolo di fronte alla squadra.

Presentazione Il facilitatore si alza e descrive il ciclo di vita da costruire, identificando chiaramente il punto in cui il ciclo di vita inizia e tutti i punti di controllo inerenti al ciclo di vita proposto.

Round 1 Definire l'inizio del ciclo di vita: Il facilitatore conduce la squadra attraverso un Checkpoint Construction per stabilire il punto di partenza / criteri di ingresso per il ciclo di vita.

- Le carte di stato successive al punto di partenza / criterio di ingresso vengono spostate a destra. Assicurarsi che rimanga uno spazio bianco chiaro tra le due serie di carte.
- Per ogni stato da raggiungere al controllo dei checkpoint controllare se è obbligatorio o solo raccomandato. In questo caso obbligatorio significa che il ciclo di vita non può essere avviato se lo stato non è raggiunto. Se lo stato non è obbligatorio ruotate la scheda di 90 gradi per indicare che è un "nice-to-have"(meglio averlo).

Round 2-N - Definire i restanti punti di controllo che compongono il ciclo di vita:

- Il facilitatore guida la squadra in un Checkpoint Construction per stabilire i criteri di uscita per il prossimo punto di controllo nel ciclo di vita.

- Le carte dello stato da raggiungere dopo il checkpoint sono spostate a destra. Assicurarsi che rimanga uno spazio bianco chiaro tra le due serie di carte.

- Riordinare le carte di stato che rappresentano il checkpoint (quelle lasciate al centro) in modo che i bordi a destra di tutti gli stati finali da raggiungere alla pietra miliare siano allineati.

- Per ogni stato da raggiungere al punto di controllo controllate se è obbligatorio o solo raccomandato. In questo caso obbligatorio significa che il ciclo di vita non potrebbe essere avviato se lo stato non è raggiunto. Se lo stato non è obbligatorio ruotate la scheda di 90 gradi per indicare che è un "nice-to-have".

Final round: Rivedere il ciclo di vita nel suo complesso e aggiustare se necessario.

2.1.6 Milestone Mapping

Questo gioco serve per visualizzare le *milestone* e formare una tabella di marcia orientativa per lo sviluppo del software.

Quando viene assegnato un pezzo importante di lavoro i team di sviluppo possono voler identificare alcune pietre miliari intermedie per aiutare a migliorare le loro stime e i loro piani. Questo è particolarmente utile quando lo sviluppo comporta alti livelli di innovazione come nuovi team, nuovi modi di lavorare, nuove tecnologie, nuovi domini di business o nuovi partner commerciali.

Il gioco è strutturato nei seguenti punti:

1. Determinare dove vi trovate giocando a "Chase the State". Il risultato finale dovrebbe essere che gli stati già raggiunti sono stati spostati a sinistra e chiaramente separati da quelli che restano da raggiungere.

2. Iniziare con la prossima milestone da raggiungere.

3. Per ogni alfa identificare lo stato in cui quell'alfa dovrebbe trovarsi quando la milestone è raggiunta:

   (a) Se lo stato non è lo stato finale dell'alfa spostate tutte le carte stato rimanenti a destra

   (b) Se lo stato alla prossima pietra miliare è lo stesso dello stato attuale sposta tutte le carte degli stati alfa a destra.

4. Riordina le carte stato che rappresentano la pietra miliare (quelle lasciate al centro) in modo che i bordi a destra di tutti gli stati finali da raggiungere alla pietra miliare siano allineati.

5. Per ogni stato da raggiungere per la pietra miliare, controlla se è obbligatorio o meno o solo raccomandato. In questo caso obbligatorio significa che la pietra miliare sarebbe fallita se lo stato non viene raggiunto. Se lo stato non è obbligatorio ruotare la carta di 90 gradi per indicare che è un "*nice-to-have*".

6. Ripeti i passi da 3 a 5 per tutte le milestone successive.

7. Rivedere l'insieme delle milestone nel suo insieme e regolare per bilanciare il lavoro da fare rispetto alla tabella di marcia proposta.

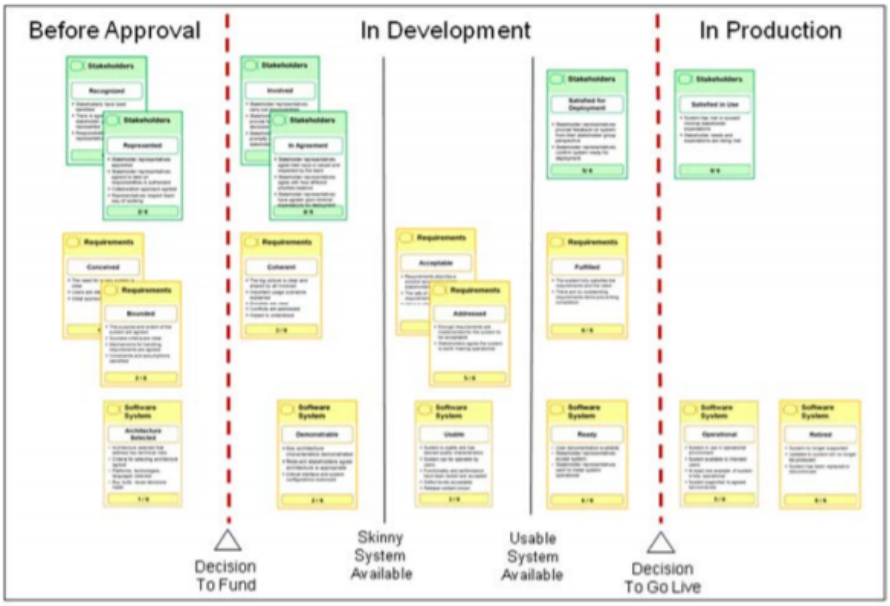

Figura 2.5: Esempio del gioco

Questa è una variazione del gioco *Objective Go* dove invece di identificare semplicemente il prossimo obiettivo la squadra usa le carte per identificare una serie di milestone da includere nel progetto o piano di rilascio.

2.1.7 Health Monitoring

Questo gioco serve per tracciare visivamente la salute dello sforzo di sviluppo, indipendentemente dalle pratiche o dal metodo utilizzato.

I diagrammi radar sono un ottimo modo per aiutare tutti a capire cosa sta succedendo, cosa è importante e come la squadra sta progredendo. Le carte stato alfa possono essere usate per creare un semplice ed intuitivo consiglio di stato che aiuta il team a monitorare i suoi progressi e la sua salute.

Questo è un gioco composito che comporta la creazione di una tavola di stato permanente come mostrato qui sotto e che coinvolge:

1. giocare a "Chase the State" per impostare lo stato iniziale

2. rivisitare regolarmente la tavola per mantenere la visualizzazione dello stato aggiornato

Figura 2.6: La state board del gioco [2]

Il gioco è strutturato nei seguenti punti:
Impostazione della scheda:

1. Usando una lavagna bianca o due fogli di carta a fogli mobili, preparate il mazzo di carte completo come mostrato sopra, con tutte le carte dello stato all'estrema destra.

2. Giocare a "Chase the State" per stabilire lo stato attuale dello sviluppo. Spostando gli stati completati a sinistra del tabellone.

   Mantenere il tabellone aggiornato

3. A periodi regolari, possibilmente come parte dello stand up quotidiano, una retrospettiva o il completamento di un lavoro, il team dovrebbe spuntare le voci della lista di controllo sulle carte e ogni volta che uno stato è completato, spostarlo a sinistra della scheda.

   Evidenziare le aree di interesse:

4. A periodi regolari, tipicamente come parte di una retrospettiva, il team dovrebbe valutare quanto ogni alfa sia in "salute" cioè come è progredito

5. Se il team pensa che un alfa sia in salute e in uno stato appropriato, adornare la carta alfa con un adesivo verde o una nota adesiva.

6. Se la squadra pensa che un alfa non sia in salute o in uno stato inappropriato, adornate la carta alfa con un adesivo rosso o una nota adesiva. Se viene usata una nota adesiva, scrivi il motivo per cui la squadra pensa che l'alfa sia malsano.

7. Se la squadra non è d'accordo se l'alfa è sano o no, allora adorna la carta alfa con un adesivo arancione o una nota adesiva. Se viene usata una nota adesiva, scrivi le ragioni per cui la squadra pensa che l'Alfa non sia sano.

   Mostrare l'obiettivo successivo:

8. Se lo si desidera, la squadra può evidenziare il prossimo obiettivo giocando a "Objective Go" usando le carte sulla lavagna e posizionando gli stati obiettivo nel mezzo tra quelli raggiunti e quelli rimanenti

2.1.8 Ultime cose sugli alfa

Questi sono i sette proposti dallo standard per usare gli alfa, nulla vieta di inventarsene altri.

Se ti stai chiedendo: "ma tutti questi alfa e carte ed Essence e tutto quanto servono veramente?", ecco una piccola risposta:

Essence aiuta una squadra a ragionare sul proprio modo di lavorare e a decidere se ci sono miglioramenti da fare.Gli sviluppatori che vengono direttamente da un programma educativo spesso sanno più di programmazione che di sviluppo software, o di lavoro di squadra e di miglioramento del loro modo di lavorare. Poiché la loro esperienza è limitata, spesso hanno bisogno di un piccolo aiuto. Gli alfa e i loro stati possono aiutare un team a ragionare sul loro modo di lavorare mentre cercano di migliorare.

Spesso, gli alfa del kernel Essence hanno bisogno di essere suddivisi in elementi più granulari per misurare i progressi giornalieri o comunque in lasso di tempo minore di quello con cui si gioca.

Alcune domande utili da fare ogni iterazione per ogni alfa per chi il facilitatore dei giochi o in generale il leader del team:

1. Cosa è andato bene durante questa iterazione e abbiamo raggiunto questo stato alfa?

2. Cosa non è andato bene durante questa iterazione, e sappiamo cosa ci impedisce di raggiungere questo stato alfa?

3. Cosa possiamo fare meglio nella prossima iterazione che ci aiuterà a raggiungere questo stato alfa?

Una piccola nota per chiarire una aspetto importante:

Anche se gli alfa sono separati, non sono indipendenti. Essi rappresentano solo visioni diverse dello stesso processo di sviluppo. Come tale, in un progetto gli stati alfa del kernel progrediscono a ondate. Dobbiamo fare delle mosse che portano il progetto da una serie di stati ad un'altra serie di stati, e l'equilibrio si ottiene facendo abbastanza su ogni dimensione per essere in grado di far progredire tutte le dimensioni. Questa progressione ondulatoria funge da riferimento per individuare le anomalie.

Ultima nota sull'utilità di Essence (nel caso ci fossero ancora dubbi):

I progetti waterfall possono cadere nell'errore di produrre un Sistema Software dimostrabile troppo tardi. I progetti Agile possono cadere nella trappola di non cercare presto il feedback e il consenso degli Stakeholder. Così, gli alfa del kernel forniscono un controllo semplice ma olistico del progresso e dello stato di salute.

2.2 Scrum con Essence

2.2.1 Cos'è Scrum

Scrum è un modello di processo molto popolare per aiutare i team a collaborare e lavorare efficacemente in modo iterativo.

Scrum è un modello di sviluppo iterativo dove ogni iterazione, o time-box, è chiamata sprint. Lo sprint è un alfa, qualcosa da osservare. Scrum guida i team a completare gli elementi di lavoro in un backlog. Questi elementi di lavoro, conosciuti come Product Backlog Items (PBIs) usando la terminologia Scrum, possono anche essere trattati come alfa.

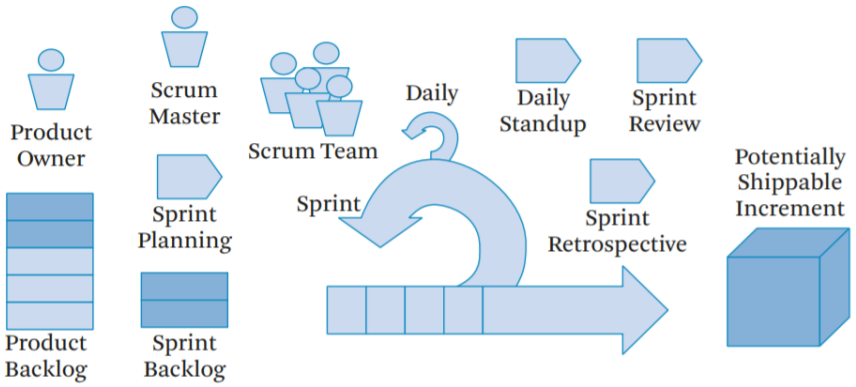

Figura 2.7: Scrum in un immagine

La Figura 2.7 definisce tutto Scrum; ora vediamo in dettaglio ogni componente.

Tutto il lavoro che il team potrebbe dover fare viene prima messo in una lista ordinata: il backlog del prodotto.

Il backlog del prodotto riporta le voci più importanti in cima alla lista. Le voci nel backlog del prodotto sono chiamate elemento del backlog di prodotto (*Product Backlog Items PBIs*)). Un PBI può essere un pezzo di un requisito, qualcosa che il team può fare per migliorare se stesso, o difetti che dovranno essere risolti.

Il cuore di Scrum è lo sprint, un periodo di tempo di lunghezza fissa, di solito da una a quattro settimane, durante il quale il team raggiunge un certo obiettivo, che include la produzione di un incremento potenzialmente spendibile del prodotto da sviluppare.

Le PBI da eseguire in uno sprint sono selezionate attraverso l'attività di pianificazione dello sprint dove il team, insieme al proprietario del prodotto (PO), si accorda sulle PBI con la massima priorità su cui lavorare nello sprint successivo. Queste PBI sono spostate dal product backlog allo sprint backlog. Questa attività è fatta il primo giorno di ogni sprint dall'intero Scrum team che lavora insieme per determinare cosa può essere consegnato e come può essere consegnato nel periodo di tempo concordato per lo sprint.Ci sono due parti nell'attività di pianificazione dello sprint. Durante la prima parte, l'PO spiega al team gli obiettivi dello sprint e i PBI che, se implementati, raggiungerebbero l'obiettivo dello sprint.

Ogni giorno durante lo sprint, il team si incontra per sincronizzare il proprio lavoro e creare un piano per le prossime 24 ore. Questo è chiamato daily scrum (oppure daily standup)) ed è limitato a 15 minuti. Al daily scrum, ogni membro del team spiega cosa ha fatto dall'ultimo incontro, cosa ha intenzione di fare oggi e cosa lo ostacola impedendogli di raggiungere l'obiettivo dello sprint. Le soluzioni ai problemi non sono discusse nel daily scrum.Un incontro separato è organizzato per approfondire i problemi

quando necessario.

Alla fine dello sprint, il team conduce un'attività di revisione dello sprint con gli stakeholder chiave per rivedere il prodotto nella sua versione corrente.In questa revisione, gli stakeholder possono anche identificare i miglioramenti del prodotto (PBI) che saranno messi nel backlog del prodotto. Alla fine di ogni sprint, il team tiene un'attività di retrospettiva dello sprint. La retrospettiva dello sprint è un'opportunità per il Scrum team di concordare miglioramenti al loro modo di lavorare, da implementare nello sprint successivo.

Dentro Scrum ci sono tre ruoli principali:

1. Product Owner(PO)

   Il PO è responsabile dell'alimentazione del product backlog basato sulla sua interazione con i clienti e gli utenti. Il PO è anche responsabile della prioritizzazione delle PBI.

2. Scrum master

   Il ruolo dello Scrum master è qualcosa di unico per Scrum. Lo Scrum master è un servant leader(), una persona che facilita le attività Scrum e motiva i membri del team a seguire le attività Scrum.

3. Scrum team

   I membri del team (cioè gli sviluppatori) sono responsabili della stima dello sforzo per implementare ogni PBI.

Scrum è essenzialmente una pratica, o meglio, un insieme di pratiche. In breve, una pratica consiste nel fare le cose in un certo modo per affrontare certi problemi, e con Scrum, si tratta di migliorare la collaborazione e le prestazioni dei team.

In questo capitolo mostriamo una versione ridotta di Scrum, chiamata *Scrum lite*. A differenza di Scrum completo Scrum lite non include una discussione sulle idee di Scrum o tutte le responsabilità di tutti i ruoli di Scrum, né includiamo tutte le caratteristiche di uno Scrum Team.

2.2.2 Scrum attraverso Essence

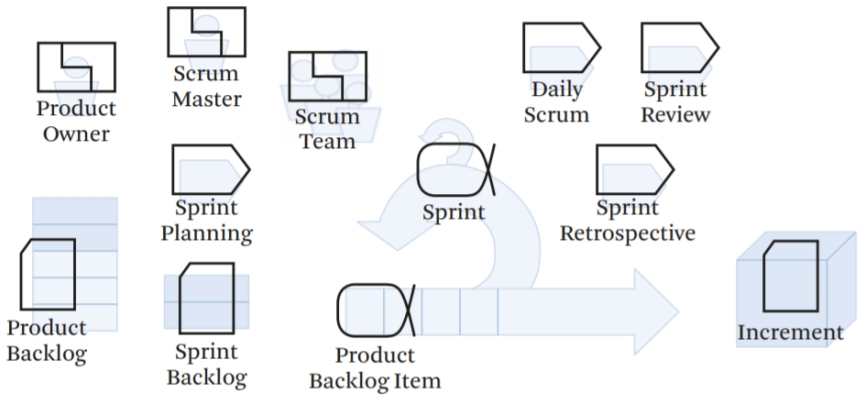

Figura 2.8: Scrum con il linguaggio Essence

Nella figura 2.8 vediamo Scrum ridisegnato tramite Essence. I ruoli PO, Scrum Master, e Scrum Team sono rappresentati come pattern(ruoli) invece Sprint Planning, Daily Scrum, Sprint Review, e Sprint Retrospective sono attività poi Product Backlog, Sprint Backlog, e Increment sono prodotti di lavoro ed infine, Sprint e Product Backlog Item sono alfa.

Nella Fig.2.9 invece vediamo un modello completo di Scrum scritto usando il linguaggio Essence.

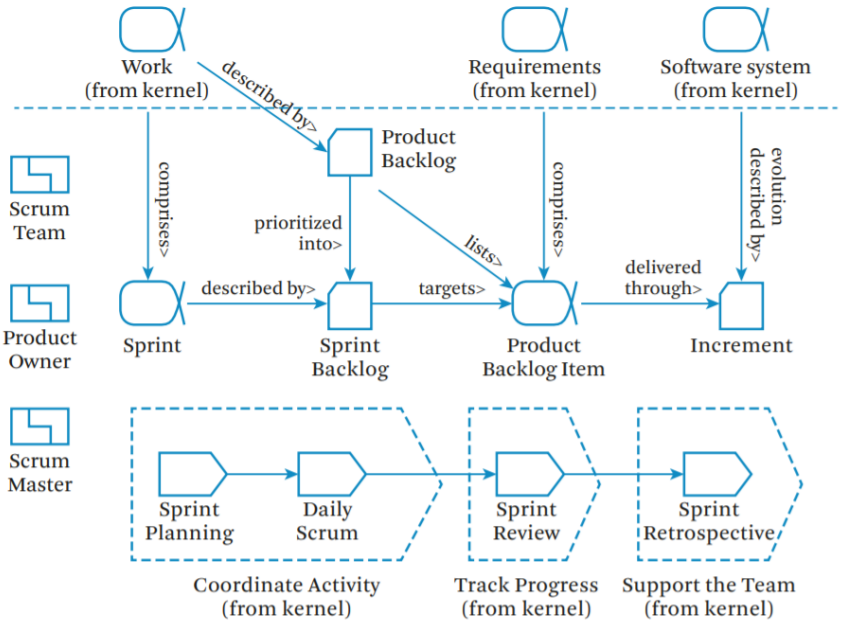

Figura 2.9: La pratica di Scrum Lite espressa nel linguaggio Essence

Un concetto importante utilizzato in Scrum è la "Definition of Done" (DoD). La DoD è una lista chiara e concisa di criteri che ogni PBI deve soddisfare perché il team la chiami completa. Un esempio di DoD potrebbe essere;

1. sufficientemente testato
2. accettato dal PO
3. codice sorgente controllato
4. i documenti associati (per esempio, i manuali d'uso) aggiornati

Il DoD deve applicarsi a tutti gli elementi del backlog.

Non discuteremo in dettaglio quali sono gli alfa e come si usano perché ne abbiamo già parlato in precedenza e non hanno nulla di diverso dagli alfa del kernel. Invece vedremo i ruoli e gli artefatti (*work product*) di Scrum.

Artefatti

I prodotti di lavoro in Scrum sono tre:

- Backlog del prodotto
- Sprint Backlog
- Incremento

Il backlog di prodotto (Product Backlog) è l'unica fonte di requisiti per qualsiasi cambiamento da fare al prodotto. Gli elementi nel backlog del prodotto sono conosciuti come PBI. C'è solo un livello di dettaglio in un backlog di prodotto:

Elementi ordinati.

Gli elementi del backlog del prodotto sono catturati nel backlog del prodotto, che può essere sotto forma di un foglio di calcolo o all'interno di qualche strumento di gestione del backlog. Sono ordinati secondo la loro priorità, in modo che quelli ad alta priorità possano essere selezionati per il prossimo sprint.

Nota: Per progetti più articolati, ci potrebbero essere più livelli di dettaglio. Per esempio, il team potrebbe voler descrivere i motivi per

Figura 2.10: La carta del prodotto di lavoro con Scrum: Product Backlog

dare priorità ai PBI, in modo che i membri possano evitare discussioni inutili.

Vediamo la carta del backlog di prodotto in Fig.2.10.

Lo **sprint Backlog** è l'insieme delle PBI selezionate per uno sprint. Include anche un piano per consegnare un incremento realizzando l'obiettivo dello sprint concordato. Uno sprint backlog rende visibile tutto il lavoro che il team di sviluppo identifica come necessario per raggiungere l'obiettivo dello sprint.

Lo Sprint Backlog comprende i seguenti livelli di dettaglio:

Obiettivi specificati. L'obiettivo dello sprint è chiaramente dichiarato e stabilisce l'obiettivo per i membri del team.

Capacità descritta. Viene stimata la quantità di lavoro che il team

può eseguire. In questo modo il team può determinare se ha troppo o troppo poco lavoro nello sprint.

Previsione di lavoro descritta. Il team è d'accordo sugli elementi del product backlog che possono essere completati entro lo sprint, così come le date obiettivo che si aspettano di completare entro lo sprint.

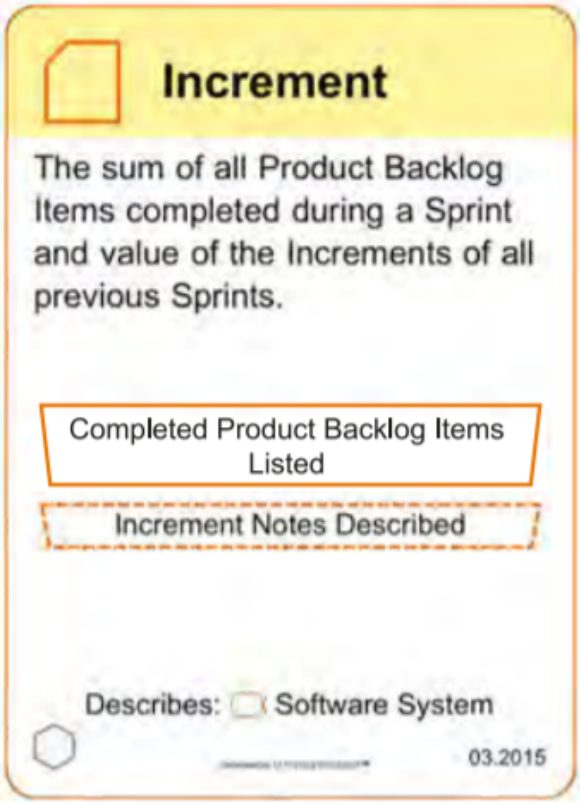

Figura 2.11: La carta del prodotto di lavoro con Scrum: Incremento

Un **incremento** è la somma di tutti gli elementi del product backlog completati durante uno sprint e quelli completati durante tutti gli sprint precedenti.

Il prodotto di lavoro Increment ha i seguenti livelli di dettaglio

PBI completati elencati. I PBI che compongono l'incremento sono chiaramente elencati.

Note sull'incremento descritte. Vengono fornite ulteriori informazioni sull'incremento, come gli ambienti in cui l'incremento può funzionare,

i problemi noti e così via. Per ambiente intendiamo quale versione di browser, quale sistema operativo e così via. Il contenuto specifico deve essere concordato dal team.

Ruoli

Scrum Lite identifica esplicitamente due ruoli, il PO e lo Scrum Master. Un ruolo è una lista di responsabilità che una o più persone accettano. Le persone che servono come PO e Scrum Master e il resto dei membri del team formano il team Scrum.

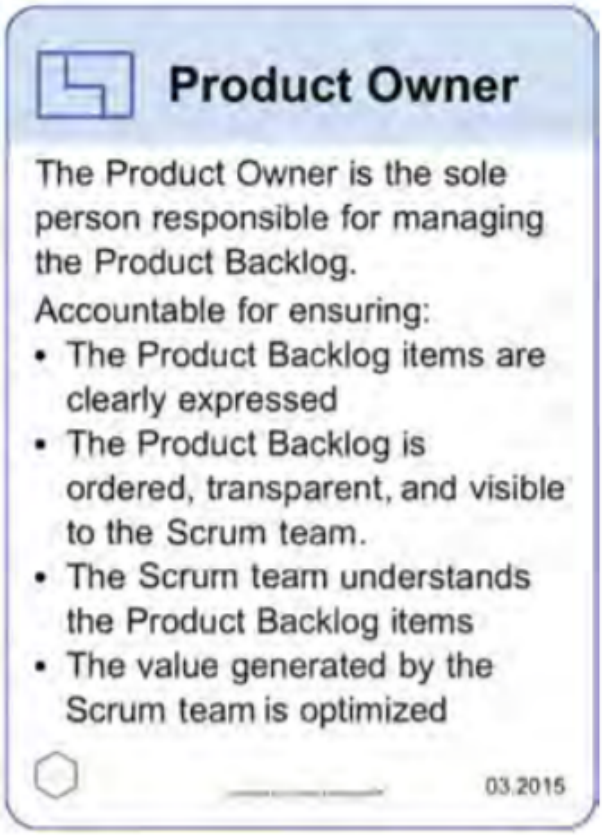

Figura 2.12: La carta del ruolo con Scrum: Product Owner

La carta in Fig.2.12 mostra che il **PO** è responsabile della gestione del backlog del prodotto, assicurandosi che ogni elemento sia chiaro ai membri del team, e assicurandosi che il backlog del prodotto sia visibile al team. La carta mostra anche che il PO è responsabile di assicurare

il valore generato dal team. Scrum è ottimizzato, il che significa che il lavoro del team fornisce valore per raggiungere l'obiettivo dello sprint.

La persona che agisce come **Scrum Master** allena il team mentre coordina le attività del modello Scrum. Quando i membri del team affrontano impedimenti, come elementi del backlog poco chiari, lui/lei lavora per rimuovere l'impedimento.

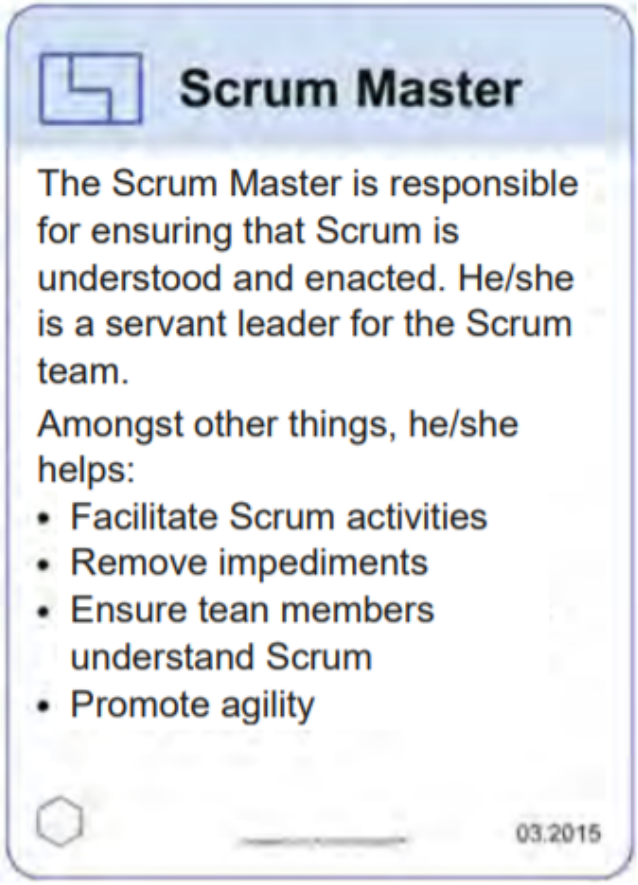

Figura 2.13: La carta del ruolo con Scrum: Scrum Master

Il **team Scrum** è composto da membri, due dei quali giocano i ruoli di un PO e di uno Scrum Master.

I team Scrum sono auto-organizzati, il che significa che nessuno al di fuori del team dice al team come raggiungere l'obiettivo di ogni sprint. Il team include persone con le competenze necessarie per realizzare tutto

il lavoro richiesto. A volte ci si riferisce a questo come ad un team interfunzionale.

Figura 2.14: La carta del ruolo con Scrum: Scurm Team

Queste carte possono essere usate dai membri del team Scrum mettendole su una lavagna o facendole portare in tasca dove possono essere facilmente accessibili come promemoria rapido delle responsabilità concordate.

Attività

Le carte attività di Scrum sono quattro:

Pianificazione dello sprint

Si tratta di decidere quali elementi prioritari del Product Backlog devono andare nello Sprint Backlog corrente(per quello sprint). Di seguito alcune domande per decidere quali elementi dovrebbero andare nello Sprint Backlog:

- Gli elementi che stiamo selezionando per questo sprint sono adeguatamente preparati e pronti per essere lavorati?
- Il team ha considerato la sua capacità quando ha deciso se può impegnarsi per gli elementi proposti da completare in questo sprint?

Il Product Backlog è la "singola fonte di verità" per tutto il team. Tutto ciò che il team potrebbe mai aver bisogno di fare deve alla fine essere aggiunto al Product Backlog.

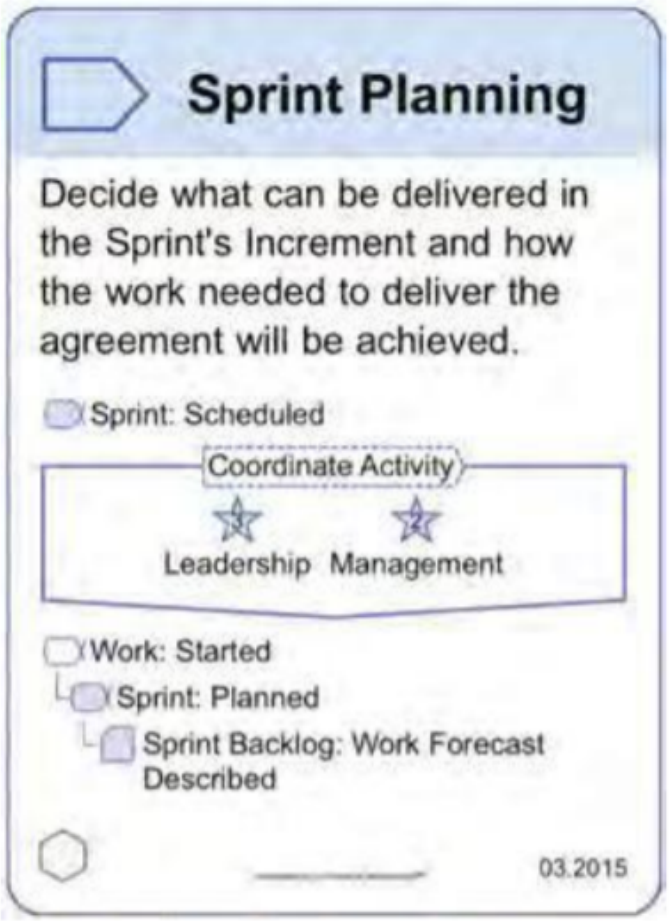

Figura 2.15: La carta attività con Scrum: Pianificazione dello sprint

Scrum quotidiano Il Daily Scrum (o Scrum quotidiano) è un'attività che i team Scrum conducono ogni giorno. Ci sono solo alcuni principi guida richiesti, come mantenere la durata della riunione entro 15 minuti,

far parlare solo gli sviluppatori, e mantenere l'attenzione sulla risposta alle tre domande principali (cosa ho fatto dall'ultimo scrum quotidiano, cosa ho intenzione di fare dopo, e quali ostacoli sto affrontando).

Il valore nel documentare queste linee guida come elementi della lista di controllo in un'attività è che servono come promemoria per il team che può aiutarli a condurre il daily scrum in modo coerente. Questi elementi della lista di controllo possono anche essere usati durante le sessioni di formazione e di coaching. Sono anche utili per addestrare i nuovi assunti.

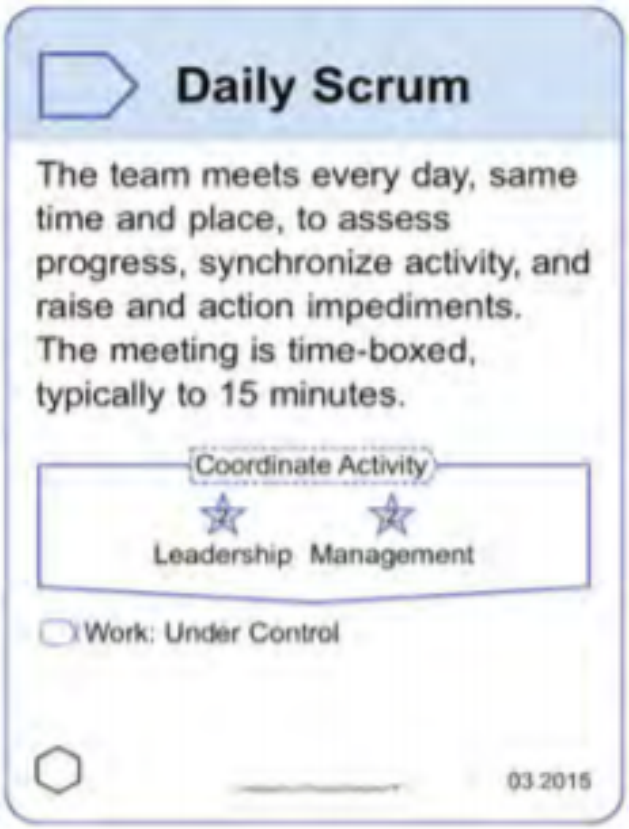

Figura 2.16: La carta attività con Scrum: Scrum quotidiano

Revisione dello sprint è una revisione del prodotto da parte degli stakeholders. Il focus di questa revisione dovrebbe essere la dimostrazione di ciò che il team ha prodotto sulla base ciò che si sono impegnati a produrre nella precedente sessione di pianificazione dello sprint.

Solo gli elementi dello sprint backlog che sono stati completati durante lo sprint sono stati dimostrati. Se qualcosa è parzialmente completato, la sua dimostrazione è rimandata allo sprint successivo, quando può essere dimostrata completamente. I team Scrum non prendono "credito parziale" per il completamento di parte di un elemento del backlog. Se gli elementi del backlog impegnati nello sprint non sono completati, il PO lo spiega durante la revisione dello sprint e spiega il piano per affrontare l'elemento mancante. La sprint review è anche un'opportunità per i membri del team di ottenere un prezioso feedback dagli stakeholder. Questo avviene principalmente alla fine della sprint review, quando il product owner chiede agli stakeholder se pensano che l'obiettivo dello sprint sia stato raggiunto.

Spesso, quando i team Scrum operano solo con pratiche tacite, la sprint review può perdere il suo focus, con gli stakeholder che sollevano problemi che non sono mai stati pianificati come parte dello sprint, o i membri del team che discutono il metodo che stanno seguendo piuttosto che il prodotto che hanno prodotto. Il valore nell'aggiungere una semplice attività di Sprint Review, o almeno l'aggiunta di liste di controllo, è che queste liste di controllo possono aiutare il team a ricordare le attività concordate relative alla sprint review.

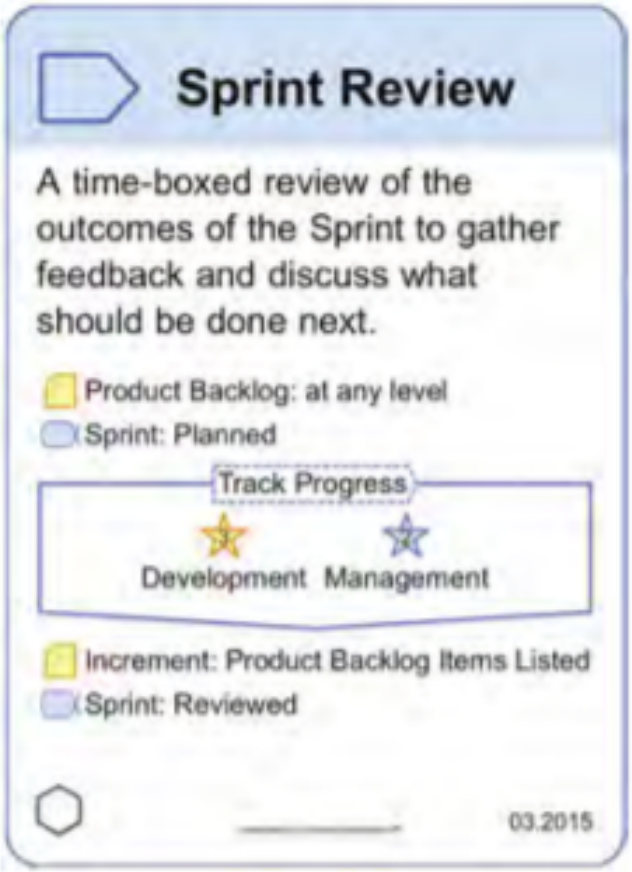

Figura 2.17: La carta attività con Scrum: Revisione dello sprint

Retrospettiva dello sprint

Lo scopo di una Sprint Retrospective è per il team di rivedere come stanno facendo il loro sforzo dalla prospettiva del loro metodo concordato, e di concordare miglioramenti al loro metodo da implementare nello sprint successivo. I risultati di questi miglioramenti possono essere taciti o espliciti, il che significa che possono o non possono richiedere modifiche alle descrizioni delle pratiche.

Il valore della Sprint Retrospective è quello di ottenere un feedback dal team di sviluppo su cosa sta funzionando bene e cosa non sta funzionando bene, e ottenere un accordo con il team su cosa possono fare diversamente durante lo sprint successivo per migliorare il loro metodo

Figura 2.18: La carta attività con Scrum: Retrospettiva dello sprint

2.2.3 Giocare con Scrum: Practice Patience

Adesso che sappiamo cos'è Scrum, come si usa ed abbiamo le sue carte, vediamo come utilizzarle tramite un gioco. Il gioco in questione si chiama Practice Patience, prima di tutto creiamo una griglia come quella in foto:

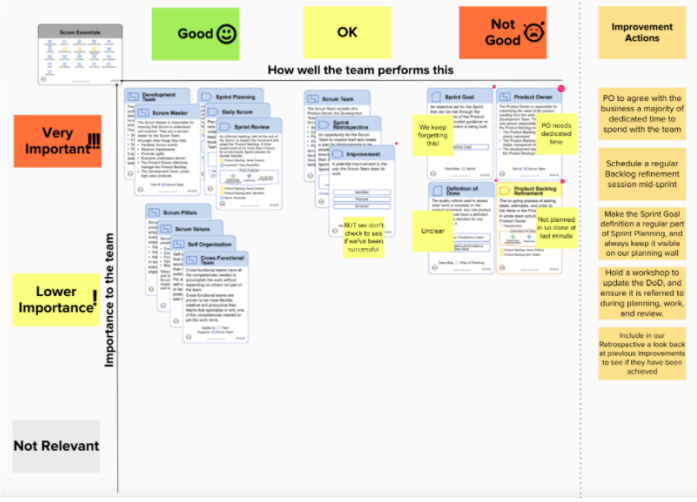

Figura 2.19: Griglia per giocare a Practice Patience

La griglia ha le seguenti caratteristiche:

- L'asse verticale rappresenta quanto è importante per la squadra
- L'asse orizzontale rappresenta quanto bene il team sente di eseguire quel concetto
- Viene creata una colonna sul lato destro per guidare le azioni di miglioramento da creare piuttosto che limitarsi a parlare dei problemi affrontati.

Come secondo step, la squadra esamina ogni carta nel mazzo e concorda dove posizionarla sulla griglia. È possibile aggiungere note a qualsiasi carta per catturare il motivo per cui è stata inserita lì per aiutare a identificare i problemi incontrati.

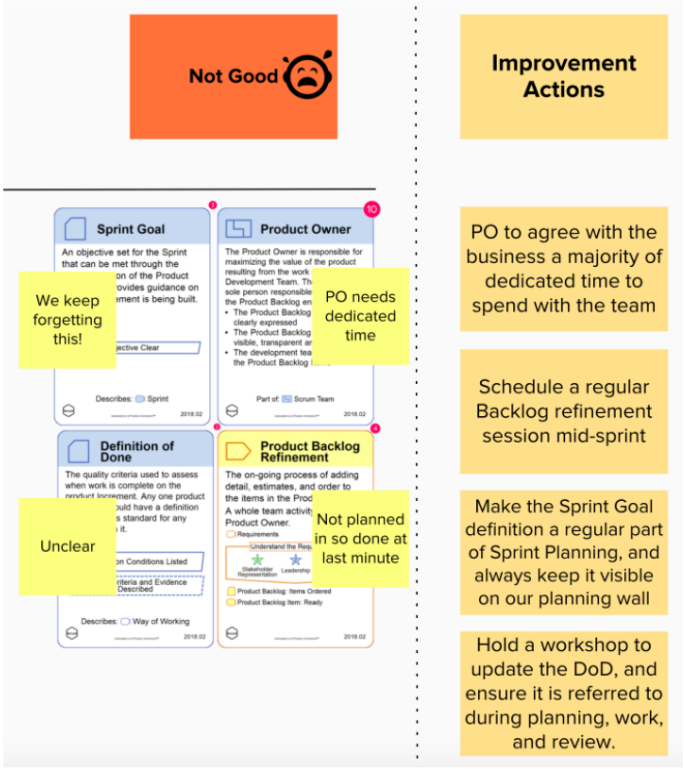

Figura 2.20: Esempio di nota sulla carta

Le carte in alto a destra sono quelle su cui la squadra deve concentrarsi in quanto considerate importanti con maggiori margini di miglioramento. Se ci sono molte carte in quell'area, il team può dare loro la priorità, magari con il voto a punti, per concentrare i propri sforzi.

Infine, il team discute la natura di ciascun problema e propone alcune azioni di miglioramento che possono essere intraprese nel successivo Sprint Planning.

Questo gioco organizzato per le retrospettiva può aiutare i team che sono abbastanza nuovi in Scrum o per i team esperti che non utilizzano Essence da un po'. Inoltre, richiede ai team di agire e pianificare i propri miglioramenti.

Per ulteriori dettagli sul gioco consultare questa pagina web [7].

2.3 La teoria dietro ad Essence

In questo paragrafo parliamo del motivo che è dietro alla realizzazione di Essence. La prima domanda da farsi è la seguente: "perché così tante discipline scientifiche hanno esplicitato la propria teoria mentre l'ingegneria del software no?" Da questa domanda ne nasce un altra: "cosa si intende per teoria?"

Una buona definizione l'ho trovata in un articolo pubblicato in Management Information Systems Quarterly [8]. Secondo l'autrice Shirley Gregor, ci sono molte definizioni, ma la maggior parte delle teorie condividono tre caratteristiche:

- cercano di generalizzare osservazioni e dati locali in una conoscenza più astratta e universale;
- hanno generalmente un interesse nella causalità (causa ed effetto);
- spesso mirano a spiegare o prevedere un fenomeno.

In dettaglio. Il primo obiettivo è quello di descrivere semplicemente il fenomeno studiato. Il secondo obiettivo è spiegare il come, il perché e il quando dell'argomento: la teoria della cognizione, per esempio, mira a spiegare i limiti della mente umana. Il terzo obiettivo è non solo spiegare ciò che è già accaduto, ma anche prevedere ciò che accadrà in seguito, ad esempio nell'ingegneria del software, il modello parametrico COCOMO cerca di prevedere il costo dei progetti software.

2.3.1 I tre argomenti

Nell'ingegneria del software non c'è una teoria così consolidata e diffusa. In questo articolo [9] del 2013, gli autori si sono provati ad immaginare il perché la comunità dell'ingegneria del software non sia interessata, ed hanno trovato tre motivazioni principali:

1. L'ingegneria del software non ha bisogno di teoria;

2. L'ingegneria del software ha già la sua teoria;

3. L'ingegneria del software non può avere una teoria.

Sempre nell'articolo viene data una risposta a tutti e tre. Riassumo la risposta al primo punto:

L'ingegneria del software non ha bisogno di teoria

L'ingegneria del software funziona bene senza teorie esplicite, quindi perché cambiare una formula vincente? Primo, l'ingegneria del software non funziona affatto bene. Numerosi rapporti sul gran numero di progetti IT falliti sono emersi da decenni ormai. Secondo, tutti i campi dell'ingegneria hanno bisogno di teoria. Per i tanti ricercatori di ingegneria del software impiegati nelle università di tutto il mondo, un ricercatore senza una teoria è come un giardiniere senza un giardino.

L'ingegneria del software ha già la sua teoria

Le teorie significative di una disciplina dovrebbero essere in grado di fornire risposte alle domande significative di quella disciplina. Considerando l'ingegneria del software, una delle questioni più dibattute riguarda la scelta del metodo di ingegneria del software. Anche se ci sono molte opinioni sull'argomento, possiamo citare pochissime teorie che tentano di rispondere alla domanda. E nella misura in cui tali teorie esistono, non sono, come in altre discipline, nominate, presentate in libri di testo o discusse in conferenze.

Lo stesso vale per altre questioni significative dell'ingegneria del software, come quale linguaggio di programmazione usare, come specificare i requisiti di sistema, e così via. Si noti che esistono molti metodi di ingegneria del software, linguaggi di programmazione e formalismi di specifica dei requisiti, ma pochissime teorie spiegano perché o predicono come

un metodo o un linguaggio sia preferibile ad un altro, in determinate condizioni.

L'ingegneria del software non può avere una teoria

Possiamo contrastare questo argomento ribadendo la stretta connessione tra ingegneria e scienza. Non è vero che non c'è teoria nella comunità dell'ingegneria software. In un certo senso, la teoria è abbondante, come ad esempio la legge di Conway, la teoria di Dijkstra dei limiti cognitivi come presentata nel classico articolo "*Go to statement considered harmful*" [10], lo stepwise refinement, e tanti altri. Ma tutte queste teorie sono piccole e la maggior parte sono casuali, proposte dall'autore ma raramente sottoposte a studi estesi, e spiegano solo un insieme limitato di fenomeni. Inoltre, la maggior parte di queste teorie non sono soggette a una seria discussione accademica. Non sono valutate o confrontate rispetto ai tradizionali criteri di qualità teorica come la coerenza, la correttezza, la completezza e la precisione.

2.3.2 Essence come teoria

Come appena detto non esiste una teoria generale predittiva ampiamente accettata dell'ingegneria del software. Essence fa un primo passo proponendo una teoria descrittiva generale e coerente dell'ingegneria del software (cioè, un linguaggio per l'ingegneria del software). Le nozioni incluse in Essence sono state progettate per catturare le caratteristiche più importanti dei vari fenomeni dell'ingegneria del software. Come teoria descrittiva, Essence può essere usata per descrivere e facilitare la discussione di future teorie predittive dell'ingegneria del software.

Essence ha fornito un terreno comune per l'ingegneria del software. Più che un semplice modello concettuale, il kernel offre:

- un quadro di riferimento per i team per ragionare sui progressi che stanno facendo e sull'efficacia dei loro sforzi;

- un quadro di riferimento per i team per organizzare e migliorare continuamente il loro modo di lavorare;
- un terreno comune per una migliore comunicazione, con misure standardizzate e la condivisione delle migliori pratiche;
- una base per un metodo accessibile e interoperabile e definizioni di buone pratiche;
- un modo per aiutare i team a capire dove si trovano durante un processo di sviluppo e cosa dovrebbero fare dopo.

Nella Fig.2.21 sottostante possiamo vedere i tre principi che stanno alla base del kernel Essence.

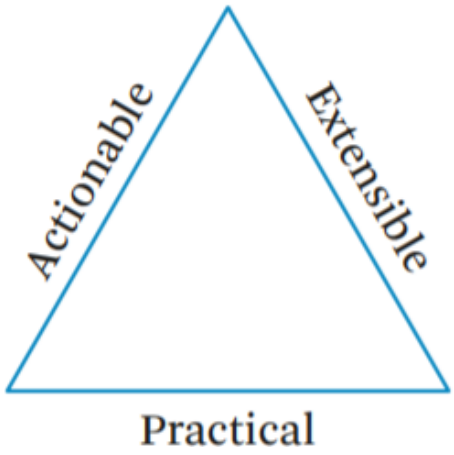

Figura 2.21: Principi guida dietro Essence

Vediamo questi principi in dettaglio:

1. Pratico

   Forse la caratteristica più importante del kernel è il modo in cui viene usato nella pratica. Il kernel costituisce un quadro di riferimento pratico e tangibile focalizzato sul supporto ai professionisti del software mentre svolgono il loro lavoro. Per esempio, il kernel è tangibile e distribuito attraverso l'uso di carte. Le carte forniscono

promemoria e spunti concisi per i membri del team mentre svolgono i loro compiti quotidiani. Fornendo liste di controllo pratiche e suggerimenti, invece di discussioni concettuali, il kernel diventa qualcosa che il team utilizza quotidianamente. Questa è una differenza fondamentale rispetto agli approcci tradizionali, che tendono ad enfatizzare troppo la descrizione del metodo rispetto all'uso del metodo.

2. Perseguibile

   Una caratteristica unica del kernel è il modo in cui sono gestite le "cose con cui lavorare". Queste sono catturate come alpha, piuttosto che prodotti di lavoro (come i documenti). Il kernel presenta l'ingegneria del software non come un processo lineare ma come una rete di elementi che collaborano: elementi che devono essere bilanciati e mantenuti per permettere ai team di progredire in modo efficace ed efficiente, eliminare gli sprechi e sviluppare ottimo software. Man mano che le pratiche vengono aggiunte al kernel, verranno aggiunti altri alfa per rappresentare le cose che o guidano il progresso degli alfa del kernel o inibiscono e impediscono il progresso.

   Il beneficio di mettere in relazione elementi piccoli coi gli elementi del kernel a più grossi è che permette il monitoraggio della salute del progetto nel suo complesso. Questo fornisce un equilibrio necessario al monitoraggio di livello inferiore dei singoli elementi, permettendo ai team di capire e ottimizzare il loro modo di lavorare.

3. Estendibile

   Un'altra caratteristica unica del kernel è il modo in cui può essere esteso per supportare diversi tipi di sviluppo. L'idea chiave qui è quella della separazione delle pratiche. Mentre il termine "pratica" è stato ampiamente usato nel settore per molti anni, il kernel ha un

approccio specifico alla gestione e alla condivisione delle pratiche. Le pratiche sono presentate come unità distinte, separate e modulari, che un team può scegliere di usare o non usare. Questo contrasta con gli approcci tradizionali che trattano l'ingegneria del software come un minestrone di pratiche indistinguibili e portano i team a mischiare il buono con il cattivo quando passano da un metodo all'altro.

Ora che abbiamo spiegato il kernel, vediamo i benefici pratici di Essence.

Man mano che Essence viene accettato dalle aziende di ingegneria del software di tutto il mondo, si sarà in grado di riutilizzare le conoscenze quando si passa dal lavorare con un sistema software a lavorare con un altro, senza dover imparare un nuovo metodo che utilizza una terminologia diversa introdotta dai suoi guru fondatori. Essence fornisce una guida per aiutare a valutare il progresso e la salute dei vostri sforzi di sviluppo, valutare le pratiche attuali e migliorare il modo di lavorare. Aiuterà anche nel migliorare la comunicazione, a muoversi più facilmente tra i team e ad adottare nuove idee.

Fornendo una base indipendente dalle pratiche per la definizione dei metodi software, il kernel ha anche il potere di trasformare completamente il modo in cui i metodi sono definiti e le pratiche sono condivise. Per esempio, permettendo ai team di mescolare e combinare pratiche da fonti diverse per costruire e migliorare il loro modo di lavorare.

I team non sono più intrappolati dai loro metodi. Possono continuamente migliorare il loro modo di lavorare aggiungendo o rimuovendo pratiche essenzializzate quando la situazione lo richiede.

I metodologi non hanno più bisogno di perdere tempo a descrivere metodi completi, possono facilmente descrivere le loro nuove idee in un modo conciso e riutilizzabile.

2.4 Conclusioni

Questo capitolo è dedicato e come utilizzare le carte in pratica. Parto spiegano i sette giochi giocabili con solo gli alfa standardizzati da Essence. Spiego come si gioca, le regole, la finalità del gioco e quando sarebbe più indicato usare un gioco o un altro. Dopodiché spiego come Essence e la metodologia Scrum si possano utilizzare insieme, mostrando le carte create appositamente per Scrum e spiegandole, spiego anche il gioco Practice Patience.

Il tutto si conclude spiegando il perché è necessaria e non solo utile la teoria di Essence e che benefici apporterà all'ingegneria del software.

Capitolo 3

La mia esperienza con Essence

In questo capitolo vedremo come io e il mio team abbiamo usate le carte Essence, per affrontare il progetto di ingegneria del software 20-21.

Il primo paragrafo spiega il progetto, in cosa consisteva, i vincoli che dovevamo seguire, il risultato.

Nel secondo paragrafo trattiamo l'uso delle carte da parte del mio team, come le abbiamo utilizzate, errori e problemi. Faremo anche un confronto con un altro team, che ha affrontato il nostro stesso progetto ma ha usato le carte in maniera diversa

3.1 Presentazione del progetto e del team

Il progetto è stato svolto in modo agile, con tre iterazioni obbligatorie e una quarta se il team ne aveva bisogno. Noi ne abbiamo fatte quattro. Ogni iterazione aveva una durata di tre settimane. La prima iterazione è iniziata il 12 ottobre, nel nostro caso il progetto avendo fatto quattro iterazioni(o sprint) è finito a inizio gennaio. Il progetto consisteva nella creazione di un raccoglitore (tracker) per i tweet. Ad esempio se si cercava un determinato hashtag (come #IngSw2020, hashtag ufficiale del progetto) il risultato doveva elencare tutti i tweet con quell' hash-

tag, visualizzati con una wordcloud e/o su una mappa. Questa era la funzionalità principale, poi c'erano altre funzionalità minori.

Figura 3.1: La pagina principale di TweeterTracker

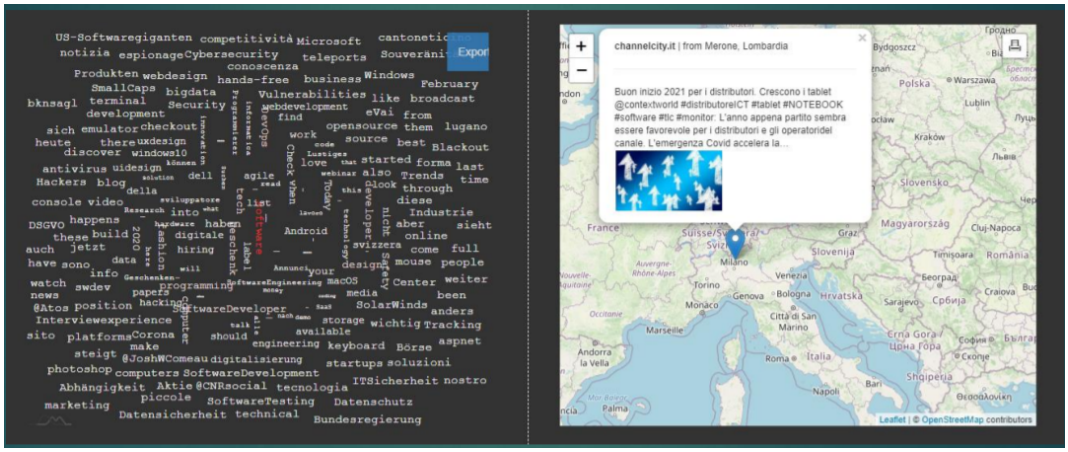

Figura 3.2: Risultato di una ricerca, visualizzazione con wordcloud e mappa

Il mio team era formato da cinque persone; tre di queste si conoscevano già e le altre due si conoscevano fra di loro, quindi possiamo dire che abbiamo unito due gruppi. Come da regolamento del progetto abbiamo usato la pratica Scrum. Il membro del team che chiameremo membro **uno** per privacy, era lo Scrum Master invece il Product Owner era impersonato dai docenti.

Siamo stati iniziati a Scrum con una spiegazione a lezione e successivamente da un gioco, che aveva come obbiettivo quello di introdurre Scrum ai team. Il gioco si chiama Scrumble ed è stato realizzato da uno studente per la sua tesi. Abbiamo eseguito lo sprint planning, la Sprint retrospective insomma abbiamo simulato giocando tutto quello che il metodo Scrum comporta.

3.2 Come abbiamo usato le carte

Le carte Essence ci sono state presentate a lezione. Sono stati presentati più che altro gli alfa e qualche metodo di utilizzarli.

Noi abbiamo usato le carte solo per fare le retrospettive. Abbiamo usato solo le carte alfa usando il gioco: Progress Poker (vedi capitolo 2.1).

3.2.1 Prima retrospettiva

Di seguito una rappresentazione della nostra prima retrospettiva con Essence:

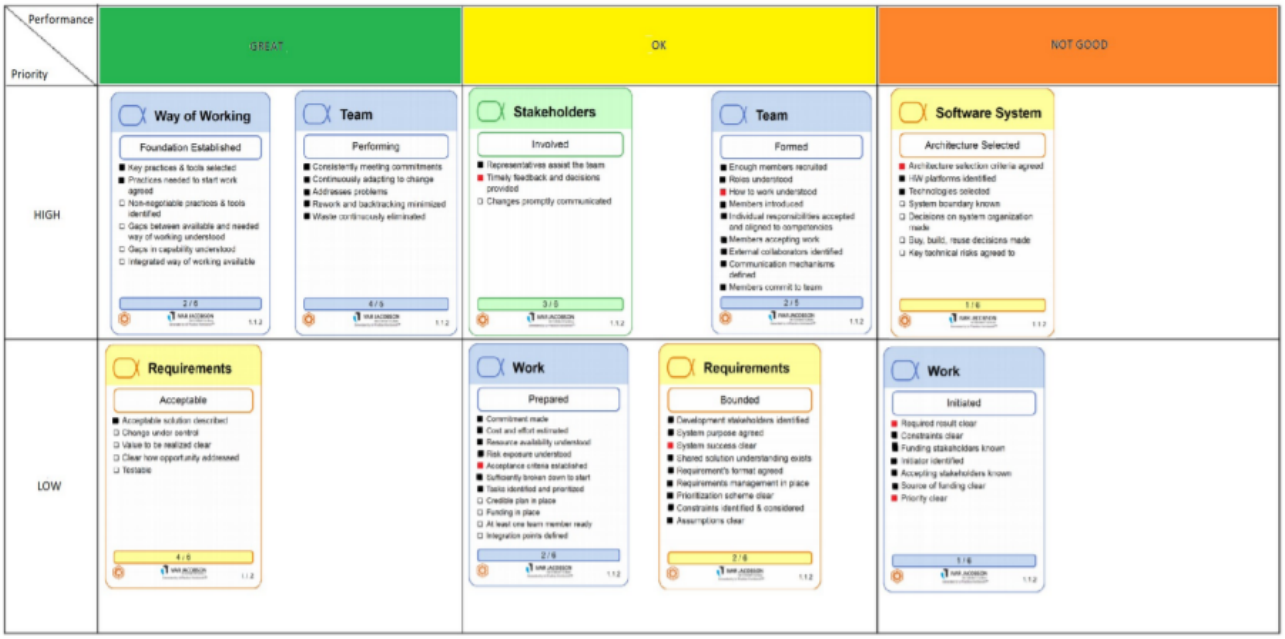

Figura 3.3: Risultato della prima retrospettiva: versione Progress Poker

Per arrivare a questo risultato abbiamo giocato Progress poker a distanza (causa Covid-19). Lo Scrum Master per ogni stato alfa leggeva i punti qualificanti e ci chiedeva se secondo noi era o meno soddisfatto quel punto o se era da mettere in rosso (cioè raggiunto ma con problemi riscontrati). Quando ci rendevamo conto che non avevamo raggiunto nessuno (o quasi) punto su quella lista, questo ci diceva che non eravamo in quello stato e quindi registravamo sul grafico (vedi figura 3.3) lo stato precedente. Nel primo sprint abbiamo impiegato molto tempo, ovvero qualche ora, complice il fatto che era il primo sprint che facevamo come gruppo e la prima volta che usavamo le carte.

Abbiamo deciso di non usare la carta alfa *Oppurtunity* perché essendo un progetto per il corso l'opportunità di business non ci riguardava, non dovendo poi commercializzare il progetto.

Notiamo nella Fig.3.3 che alcune carte sono state usate in modo errato, ad esempio le carte Team, Work e Requirements risultano sia in uno stato che in un altro, cosa non ammessa dalle regole.

Un altro errore che riscontro è l'uso dei colore per spuntare gli obbiettivi, si dovrebbero leggere così: le caselle spuntate di nero sono progressi raggiunti nello sviluppo del progetto invece le caselle spuntate in rosso rappresentano problemi riscontrati.

Sappiamo che uno stato alfa non ha mezze misure: può essere raggiunto totalmente oppure non viene segnato come raggiunto, ma anche se considerassimo questa piccola variante (cioè segnare stati come raggiunti ma colorando una o più caselle di un colore per indicare che quel punto non è stato soddisfatto), sarebbe comunque un errore segnare che una carta alfa è sia in uno stato che contemporaneamente in un altro.

Le righe indicano le priorità invece le colonne indicano la prestazione. Le colonne indicano il grado di ottenimento per quel singolo stato: in verde un grado ottimo, in giallo buono (cioè con qualcosina da migliorare), in arancione un grado in cui sono stati riscontrati dei problemi durante lo sviluppo.

La suddivisione delle carte in tre colori principali non è inutile anzi è un prezioso aiuto, così visivamente con solo uno sguardo si sa già da cosa iniziare a lavorare nel prossimo sprint. In questo caso sicuramente la prima cosa in cui si deve iniziare a lavorare è il *software system* in quanto è nella casella arancione (non bene) e ha una priorità più alta rispetto alla carta *work* anch'essa presente nel colore arancione.

Per capire meglio il significato delle carte Essence del primo sprint riporto quanto scritto (basandosi sulle carte) dal team dopo la prima iterazione:

> "Come possiamo notare dallo schema delle carte Essence, alla fine del primo sprint non avevamo completamente capito che tipo di prodotto avremmo dovuto consegnare. Avevamo iniziato a definire i primi *requirements*, che abbiamo messo come priorità *low* perché in quel momento avevamo priorità maggiori

per altri aspetti. Infatti alla fine del primo sprint non avevamo ancora chiaro quale linguaggio di programmazione utilizzare e nonostante avessimo capito a livello teorico le nuove pratiche da utilizzare, non eravamo sicuri su come metterle in pratica efficacemente."

Ora vediamo un confronto con il gruppo tre, per come hanno utilizzato le carte. Questa immagine è tratta dal loro report [1]:

[1]Report team 3,Guazzaloca. Guidi. Liso. Lorenzoni. Marzolo

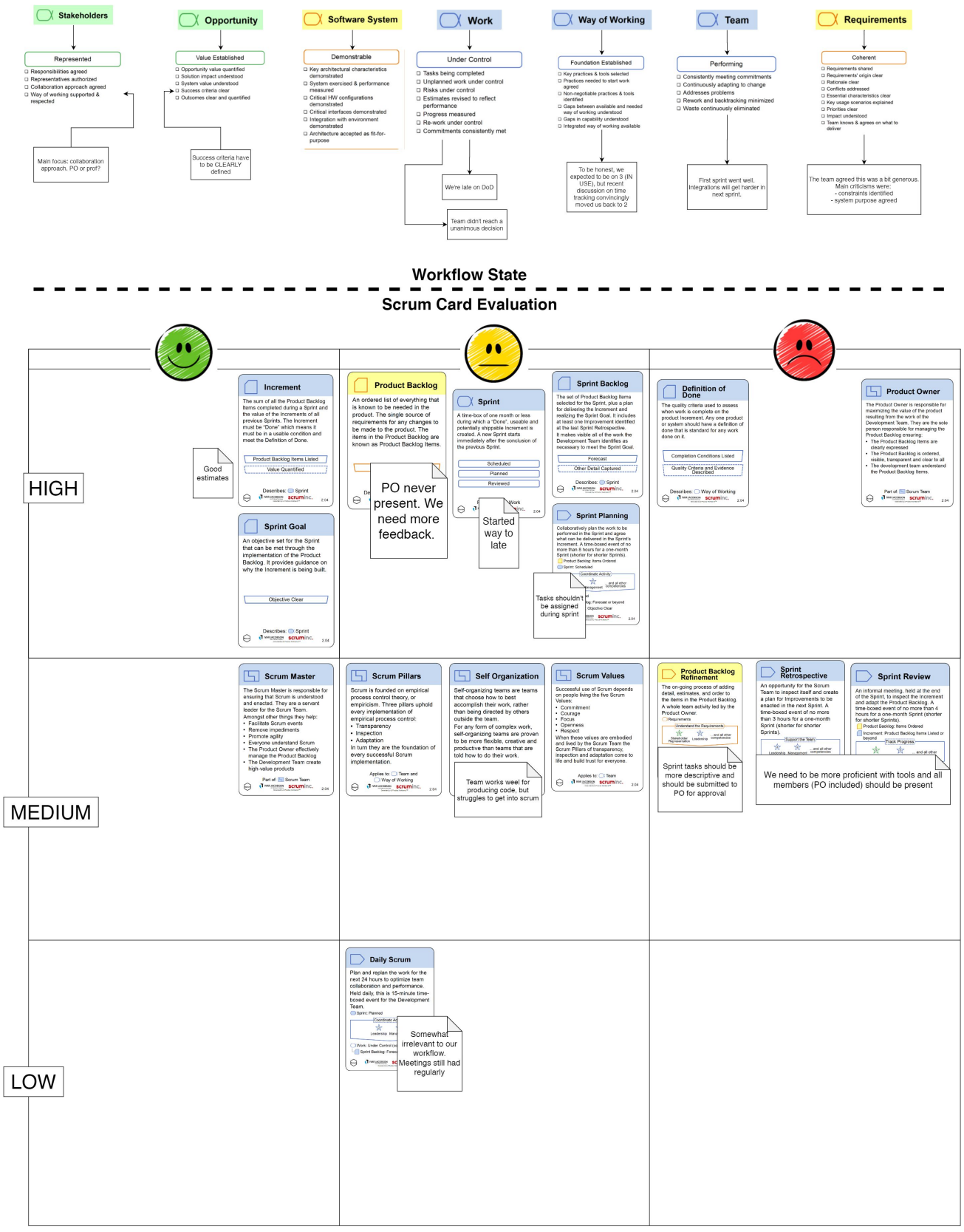

Figura 3.4: Risultato della prima retrospettiva del team 3

Vediamo subito che c'è una notevole differenza, prima di tutto per la parte inferiore dell'immagine hanno usato il metodo Practice Patience. Spiegato nel capitolo precedente 2.2.3, è un gioco che fonde le carte Essence con l'utilizzo della pratica Scrum.

Come seconda importante differenza è che hanno utilizzato tutte le tipologie di carte Essence e non solo le carte alfa, e che hanno un grafico simile al mio team ma con la piccola differenza che le loro righe sono tre e non due.

Vediamo ora il loro metodo, notiamo che in alto ci sono le sette carte alfa ed i loro rispettivi stati. Hanno raggiunto questi stati tramite una discussione, ed hanno aggiunto una nota sui punti in cui ritenevano necessario per ricordarsi di eventuali problemi o in generali di appunti, come ad esempio la nota che dice che erano in ritardo sulla "definition of done", utile per lo sprint successivo.

Nella parte inferiore, come già detto, hanno usato il metodo Practice Patience: in breve la squadra esamina ogni carta nel mazzo e concorda dove posizionarla sulla griglia. È possibile aggiungere note a qualsiasi carta per indicare il motivo per cui è stata inserita lì per aiutare a identificare i problemi incontrati. In questo modo hanno ottenuto la figura che abbiamo appena visto.

Confrontando l'uso delle carte fra il team otto e il team tre, possiamo dire che il team tre ha usato un metodo più sofisticato e secondo me anche più corretto in quanto utilizza anche le carte di Scrum e non solo gli alfa. Questo ha permesso al team di capire meglio le dinamiche di Scrum e i problemi del team durante lo sviluppo. Di contro possiamo ipotizzare che hanno impiegato più tempo di quanto non abbia fatto il team otto con il suo metodo.

Questo ci fa capire che nonostante le carte non siano state utilizzate nella maniera più corretta dal team tre hanno comunque sortito un certo

effetto, cioè: far discutere il team, esplicitare alcuni problemi (quindi scrivendoli) e di conseguenza creare un qualcosa per tenere traccia dei progressi e dei vari problemi durante lo sviluppo del progetto.

In sintesi l'utilizzo delle carte è discutibile, non le abbiamo usate benissimo in questo primo sprint ma neanche eccessivamente male.

3.2.2 Seconda retrospettiva

Vediamo il **secondo sprint**:

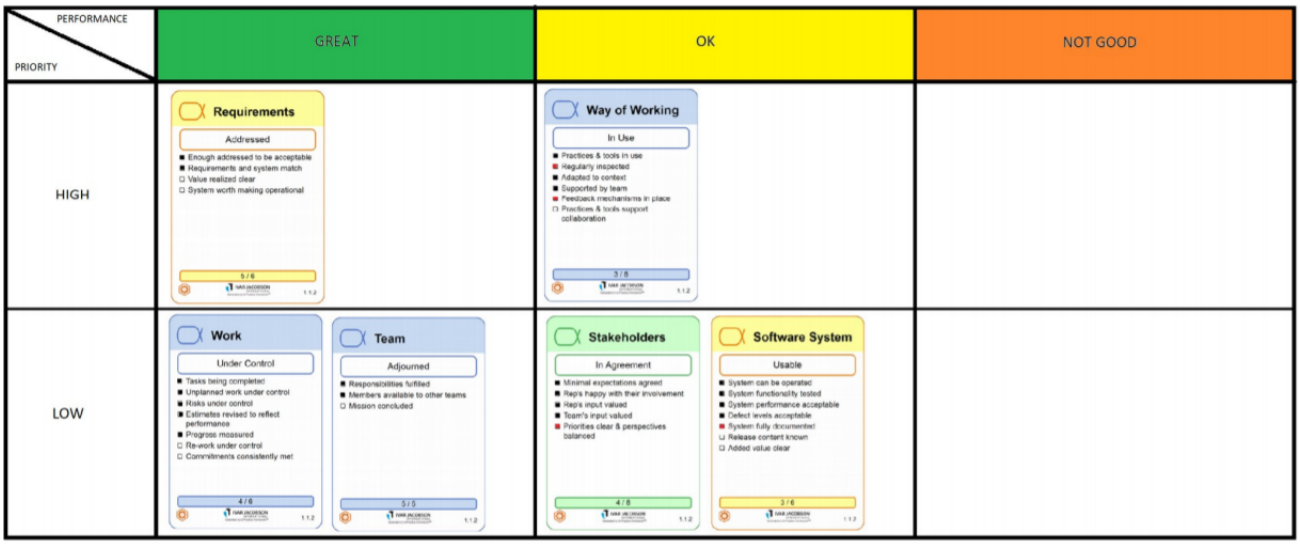

Figura 3.5: Risultato della seconda retrospettiva

In questa seconda retrospettiva non ci sono carte di stati alfa doppi: questo indica che abbiamo usato le carte un po' meglio del primo sprint. Come possiamo notare continuiamo a non usare la carta *opportunity*, e abbiamo continuato ad usare il metodo delle caselle colorate. Riporto quanto scritto dal team riguardante questa retrospettiva:

> Alla fine del secondo sprint, avevamo risolto i problemi più urgenti evidenziati durante lo sprint review del primo. Ci siamo invece accorti, alla fine del secondo sprint, che non avevamo un sistema di feedback realmente adeguato, dato che gli aggiornamenti sull' avanzamento avvenivano solo durante le riunioni di

gruppo, che però non avvenivano giornalmente. Gli aggiornamenti avvenivano per la maggior parte su chat come ad esempio sul gruppo Whatsapp.

Come possiamo notare, le carte anche se ancora una volta non usate nel modo migliore hanno assortito il loro effetto, ed il fatto di dare una priorità alle cose tramite le carte ha funzionato. L'effetto delle carte è stato quello di far comunicare il team, e fargli rendere conte dei progressi fatti e da fare. Questo dimostra che forse non c'è un metodo giusto e una sbagliato per usare le carte, dato che comunque il risultato voluto dalle carte è stato comunque ottenuto, ma sicuramente c'è un metodo più giusto e uno meno giusto.

3.2.3 Terza e quarta retrospettiva

Ora vediamo le carte della retrospettiva relativa allo sprint tre e quattro e poi guardiamo e commentiamo la riflessione del team sulle carte.

Figura 3.6: Risultato della terza retrospettiva

Vediamo che ancora una volta l'impiego di sei carte - invece di sette -, e che quasi tutti gli stati alfa sono progrediti, ad eccezione della carta *team* che però aveva già raggiunto 5 su 5 nella retrospettiva precedente.

Infine vediamo il quarto ed ultimo sprint:

Figura 3.7: Risultato della quarta retrospettiva

Per qualche motivo abbiamo deciso stavolta di usare la carta *opportunity* e di togliere la carta *software system*. Osserviamo anche che tutte le carte sono all'ultimo stato alfa. Questo potrebbe essere visto come una forzatura, in quanto secondo me è stato fatto perché essendo all'ultimo sprint il team aveva supposto che le carte dovevano essere tutte nell'ultimo stato. Ovviamente anche questo è stato un errore, le carte vanno usate sempre in modo veritiero anche alla fine di un progetto.

3.2.4 Conclusioni del team

Durante la fase di esposizione e di interrogazione sul progetto, il docente ci fece una domanda: "Riusereste le carte Essence in un prossimo

progetto?". Riassumendo alcuni punti di vista comuni, la risposta fu no. La principale motivazione fu che il metodo necessitasse di tanto tempo e che non avevano capito da subito come utilizzarle.

Col senno di poi evinciamo che con un corretto utilizzo delle carte (cioè con un metodologia migliore, usando il gioco *chasing the state* e solo in caso di dubbio giocando poi a *progress poker*, come esempio, probabilmente la risposta sarebbe stata un sì.

3.3 Conclusioni

In questo capitolo ho raccontato la mia esperienza con le carte Essence. Ho da prima spiegato che tipo di progetto era e com'era strutturato. Successivamente ho mostrato come io e il mio team abbiamo utilizzato le carte mettendoci a confronto con un altro gruppo, analizzando vantaggi e svantaggi ed errori.

Capitolo 4

La mia proposta: essenzializzare GitLab

In questo capitolo vedremo cos'è GitLab, a cosa serve, le sue principali funzionalità. Vedremo successivamente la creazione delle carte Essence di GitLab ed il loro utilizzo e la motivazione.

Tratterò principalmente di come usare GitLab in team, non dei comandi git. Per scrivere il primo paragrafo ho letto il libro *Git For Teams* [11]; tutte le figure del paragrafo 4.1 sono prese da questo libro.

4.1 Introduzione a GitLab come strumento di team

GitLab è una piattaforma web open source che permette la gestione di repository Git e di funzioni *trouble ticket*. La sua principale funzione è quella di versionamento del codice per i team di sviluppo.

Uno strumento di versionamento del codice è un sistema che permette di gestire le varie versioni del software che si sta scrivendo. Una delle funzioni principali e più importanti è il *branching*: consiste nella la gestione dei branch, cioè dei rami di sviluppo per fare in modo che si possa scrivere codice separatamente (ad esempio da due persone diverse) e in un secondo momento unire due parti di codice tramite un'operazione chiamata *merge* (si può fare anche tramite *rebase*).

Ci sono tante piattaforme che usano git. Io ho scelto di parlare di GitLab perché è quella che abbiamo usato nel progetto di Ingegneria del software 20-21. Nel prossimo capitolo vedremo come il mio team ha usato GitLab durante il progetto.

4.1.1 Licenze di distribuzione & scelta del modello di collaborazione

Ogni progetto (specialmente se open source) ha bisogno di decidere quali delle tante licenze disponibili adottare. Vediamo brevemente alcune fra le licenze di distribuzione più conosciute:

- licenza MIT - permette agli sviluppatori di fare qualsiasi cosa vogliano con il codice, purché citino l'attribuzione agli autori originali, e non li ritengano responsabili del software.
- licenza Apache è simile alla licenza MIT, ma garantisce anche esplicitamente i diritti di brevetto dagli autori, e richiede una nota di modifica che descriva come il codice derivato cambia dalla versione originale.
- GNU General Public License (GPL), è una licenza copyleft, favorevole alla condivisione, che richiede a chiunque distribuisca il codice o un suo derivato di rendere il sorgente disponibile sotto gli stessi termini.
- Creative Commons questa licenza permette di concedere i diritti di ridistribuzione, con o senza modifiche, per uso commerciale o non commerciale.
- Si può non scegliere una licenza di distribuzione, tuttavia questo segnala che non sei interessato a che altri usino il tuo lavoro senza cercare un permesso esplicito.

Per scegliere al meglio fra queste licenze si può visitare questo sito creato da GitHub stesso: [12].

Modelli di accesso

Git non ha la capacità di controllare l'accesso. Invece permette a qualsiasi sviluppatore un accesso completo in lettura/scrittura al repository. Al livello più grossolano, si limita questo controllo attraverso i controlli di accesso. Io sviluppo sulla mia macchina, alla quale voi non avete accesso, e quindi non potete cambiare il mio repository. Non appena mettiamo il repository in un luogo condiviso, come un server centralizzato di hosting del codice, dobbiamo accordarci su come gestire il nostro accesso al repository.

Sono tre i modelli più famosi, vediamoli:

1. **Modello di collaborazione disperso** Per condividere il lavoro con gli altri gli sviluppatori creavano un file patch usando il programma diff. Scrivevano poi un'email al gruppo di discussione, allegando il loro file di patch come mostrato nella Figura 4.1. Per studiare i cambiamenti proposti, i membri della mailing list scaricherebbero il file patch allegato e lo applicherebbero al loro codice locale, usando il comando patch del loro sistema.

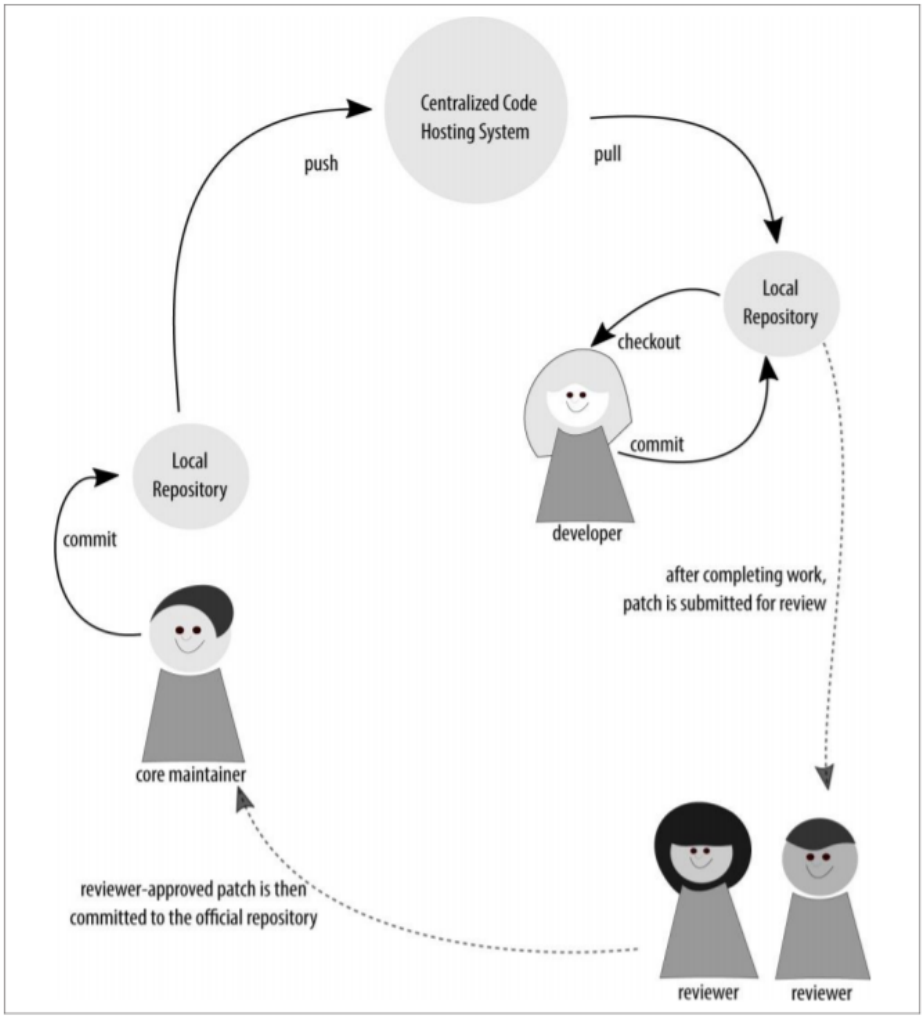

Figura 4.1: Il processo di revisione della comunità con le patch

Vediamo vantaggi e svantaggi di questo modello,**vantaggi:**

- Non c'è bisogno di usare uno specifico sistema di controllo di versione a livello locale perché il file di patch non richiede uno specifico software di controllo della versione da installare localmente.
- Gli sviluppatori possono facilmente rivedere le modifiche proposte dalla comodità tramite la loro applicazione di posta elettronica
- Questo modello incoraggia il pensiero collaborativo. Se dovete mandare una mail a un gruppo di persone ogni volta che fate

un cambiamento, è più probabile che vi assicuriate che tutto sia giusto in modo da evitare l'imbarazzo di dover cambiare "ancora un'altra cosa"

- Caricare le modifiche proposte in un sistema che non è il sistema di hosting del codice impone una procedura di revisione tra i partecipanti al progetto software. In altre parole, come sviluppatore, non posso semplicemente caricare le mie modifiche nel repository principale; devo annunciare il mio lavoro completato e aspettare che qualcun altro lo unisca al resto del sistema.

Svantaggi

- è più difficile spacchettare la storia di chi ha fatto quali modifiche se sono coinvolte più persone
- Il team dovrà aderire ad una politica di formattazione delle patch (firmate o meno), e ad uno stile dei messaggi di commit.

Per la maggior parte dei progetti contemporanei questo modello non è appropriato. Un approccio più moderno a questo modello è quello di usare *fork* o *clone*, su un singolo sistema di hosting del codice.

2. **Modello con repository di contribuzione co-locato** Su un sistema co-locato, il progetto "a monte" mantiene il controllo completo su chi è autorizzato a scrivere sul repository primario del progetto. I singoli collaboratori fanno un clone o fork del progetto nel loro repository sul sistema di hosting del codice. I collaboratori apportano modifiche alla copia, e poi inviano le loro modifiche richieste sotto forma di richiesta di merge o richiesta di pull, come mostrato nella figura 4.2. Questo modello si addice ai progetti open source.

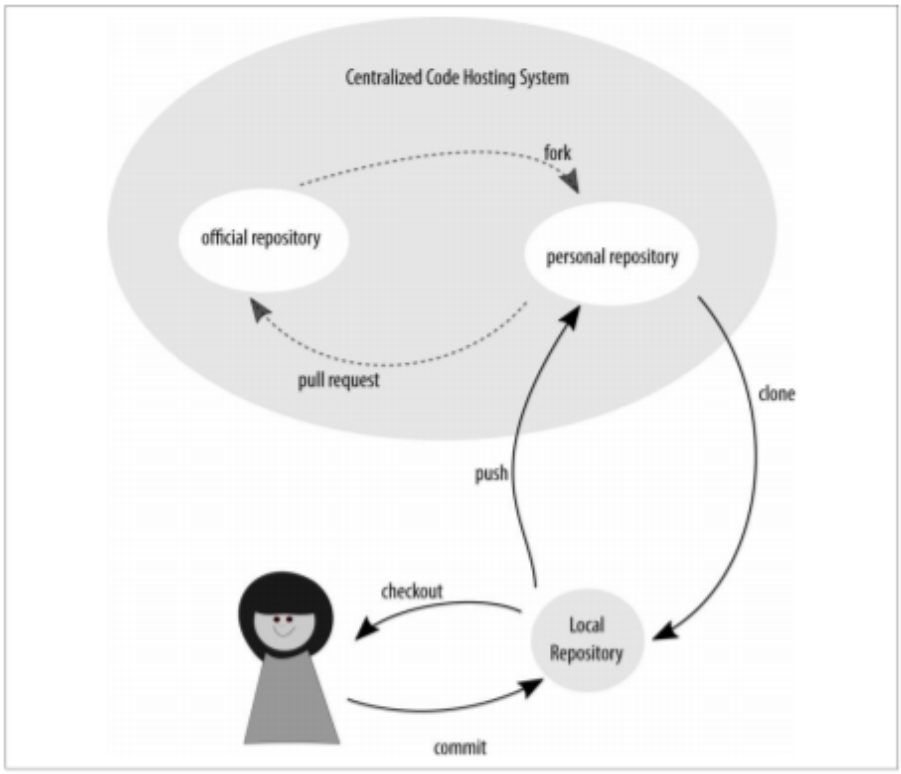

Figura 4.2: Creare una catena di repository clonati

Si crea un secondo clone del repository, questa volta dal repository biforcato alla stazione di lavoro locale. Questo crea effettivamente una catena di cloni da una copia all'altra. Mantenere tutti i repository sincronizzati richiede un po' di lavoro; tuttavia, sono molti meno comandi da memorizzare che lavorare direttamente con le patch.

Tipicamente, il primo repository nella catena può essere alterato solo da alcuni committer principali che possono aggiungere nuovi commit al repository, o unire rami. La maggior parte delle persone che lavorano al progetto, invece, lavoreranno da un clone locale del repository. In questo repository locale, clonato, ciascuno avrà un controllo totale su ciò che accade. Possono aggiungere nuovi rami, aggiungere nuovo codice, e condividere le modifiche proposte con gli altri usando il comando pull, il loro lavoro al loro clone pubblico del repository principale. Una volta che che il lavoro arriva sul

clone pubblico, i contributori possono sollecitare un feedback sul loro lavoro. Una volta che il lavoro è stato completamente rivisto e testato dalla comunità, i programmatori possono fare una richiesta di merge o una richiesta di pull dal loro clone pubblico al repository principale.

3. **Modello di manutenzione condivisa** In questo modello occorre una fiducia assoluta tra i membri del team. Si presume che il codice sarà controllato e verificato prima di essere inviato al ramo principale del progetto, e che, generalmente, gli sviluppatori siano affidabili. In questo modello, il lavoro è fatto localmente da ciascuno sviluppatore prima di essere messo nel repository condiviso per il progetto. Quando si lavora con un team, come mostrato in Fig.4.3, questo è spesso il punto di partenza: con un singolo repository condiviso in cui tutti hanno accesso condiviso in scrittura.

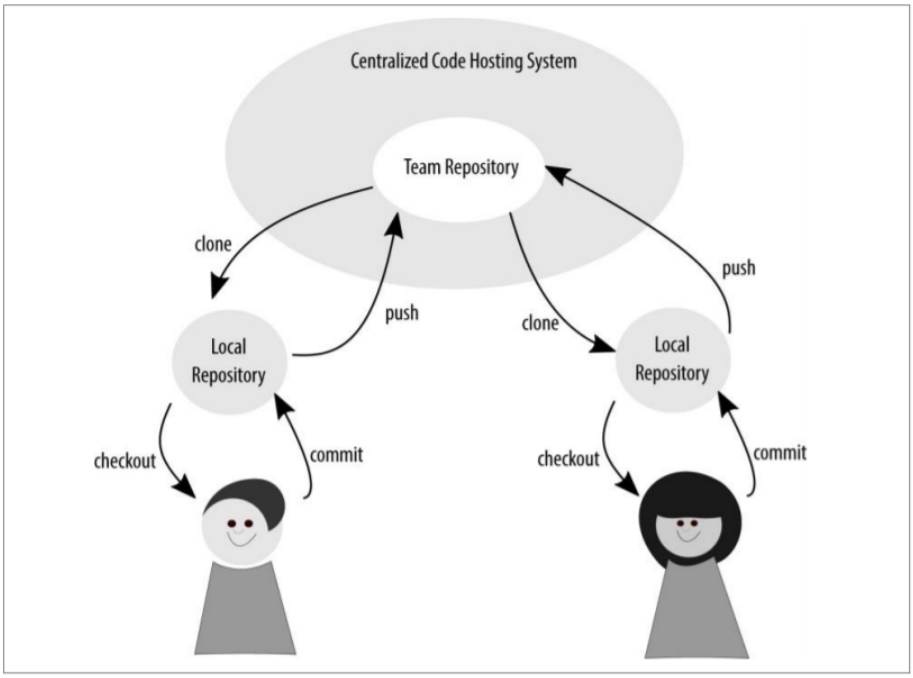

Figura 4.3: Tutti nel team hanno accesso in scrittura al repository centrale dal proprio repository locale

Git non ospita permessi e invece si affida ad altri sistemi per garantire o negare l'accesso in scrittura ad un repository. Se hai bisogno di impedire alle persone di caricare il loro codice su un repository condiviso, devi usare il controllo di accesso del sistema host per farlo (GitLab).

4.1.2 Scelta della strategia di Branching

Nel controllo di versione, un ramo (*branch*) è un modo per pensare separatamente e in parallelo su come un pezzo di codice potrebbe evolvere. Un ramo potrebbe essere creato con l'intenzione di contribuire al lavoro esistente, o potrebbe essere creato con l'intenzione di mantenere il lavoro separato. La strategia di ramificazione che si usa dipende dal processo di gestione dei rilasci. Le ramificazioni permettono di cambiare i file che sono visibili nella directory di lavoro del progetto, e solo un ramo può essere attivo alla volta. La maggior parte delle strategie di ramificazione separano il lavoro nel progetto per idee grossolane. Come per esempio separarlo per versioni, versione 1, versione 2 ecc.

Una *convenzione* è uno standard concordato per come le cose vengono fatte di solito. Come sviluppatori, le convenzioni ci permettono di prendere rapidamente i modelli di come un progetto software viene eseguito, e integrare il nostro lavoro senza interrompere il flusso per gli altri membri del team. Una convenzione documentata rende l'onboarding più facile, sia per il nuovo arrivato che per gli altri membri del team, che ora devono sottrarre meno tempo al loro lavoro per aiutare la nuova persona.

Per questi motivi è sempre meglio (anche se non obbligatorio) adottare una convenzione di branching. Vediamo quattro fra le più comuni, e per ognuna i suoi vantaggi e svantaggi:

1. **Sviluppo del ramo principale**

La strategia di ramificazione più facile da capire è il metodo della ramificazione principale. In questa strategia, ci sono meno rami con cui lavorare. Gli sviluppatori stanno costantemente caricando il loro lavoro in un singolo ramo centrale, che è sempre in uno stato pronto per il deployment. In altre parole, il ramo principale del progetto dovrebbe contenere solo codice testato e non dovrebbe mai essere interrotto. Questo metodo può andare bene se si lavora da soli o su progetti particolarmente piccoli, scegliendo di caricare tutto il codice nel ramo di default, come vediamo in figura 4.4.

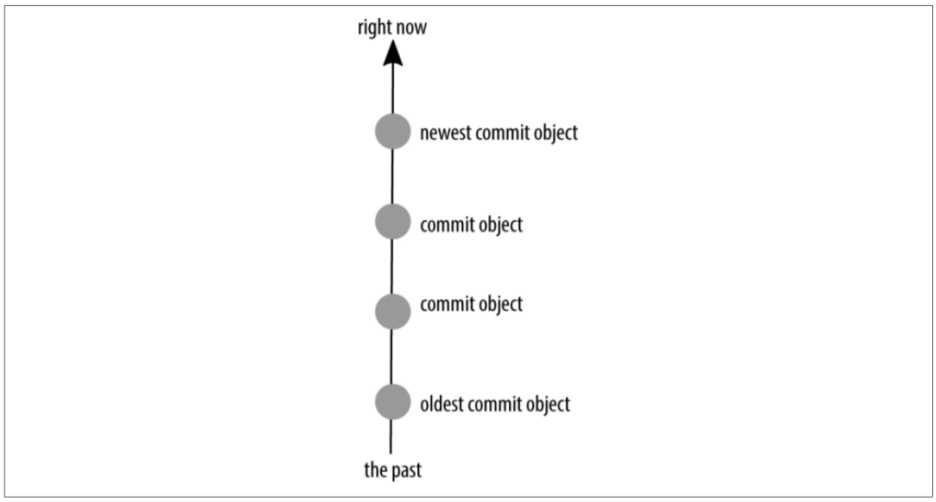

Figura 4.4: Sviluppo del ramo principale: memorizzazione di tutti i commit in un singolo ramo

Man mano che il progetto matura, ci sarà sempre più da pensare, e diventerà più difficile tenere traccia delle idee. Potrebbero aggiungersi persone al progetto, ed useranno i propri rami in maniera indipendente. La figura 4.5 si complica.

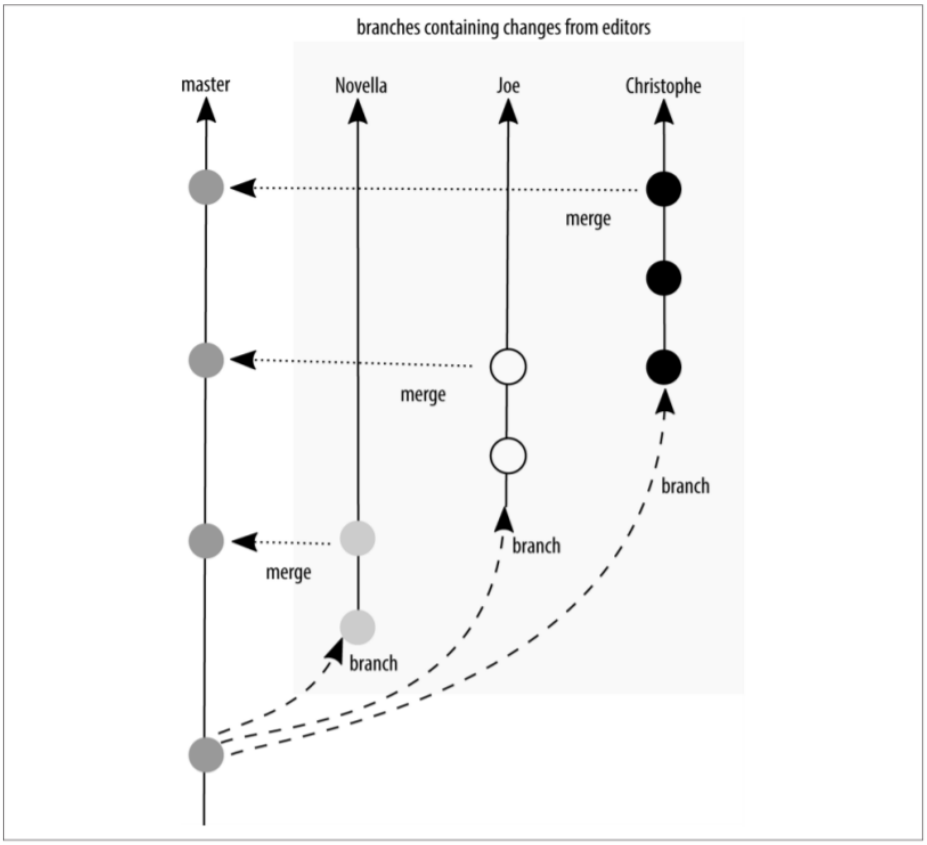

Figura 4.5: Sviluppo mainline con ramificazione: i rami separano il lavoro contribuito da più persone

Ha senso che il team integri il codice in un ramo centrale regolarmente, ma distribuisca a persone diverse il codice solo occasionalmente. Non appena si inizia a raccogliere il codice, è necessario fare una distinzione tra ciò che si ha in locale e ciò che viene utilizzato sul server di produzione. Ci sono diversi vantaggi nell'usare una strategia di ramificazione che incoraggia l'integrazione regolare del codice:

- Non ci sono molti rami in tutto il progetto. Questo si traduce in meno confusione su dove sia sparito un cambiamento
- I commit che vengono fatti nel codice principale sono relativamente piccoli. Se c'è un problema, dovrebbe essere relativamente veloce annullare l'errore.

- Ci sono meno correzioni di emergenza, perché ogni codice che viene salvato nel ramo principale è pronto per essere distribuito. I deploy possono spesso essere stressanti per gli sviluppatori, in quanto mentre il codice va in produzione aspettano di sentire la risposta degli utenti. Con piccoli aggiornamenti frequenti, questa procedura diventa pratica e infine automatizzata al punto che dovrebbe essere quasi invisibile per l'utente finale.

Ci sono anche degli svantaggi nell'usare questa strategia:

- Il presupposto è che il ramo principale contenga codice pronto per il deployment. Se il team non ha un'infrastruttura speciale di test, può essere rischioso assumere che il nuovo codice non romperà nulla, specialmente quando col tempo il progetto diventa più complesso.
- La nozione di deployment è più appropriata per il codice che viene caricato automaticamente sul dispositivo dell'utente (per esempio, un sito web). È meno appropriato per il software che deve essere scaricato e installato. Mentre gli aggiornamenti che risolvono i problemi sono i benvenuti.
- Uno dei modi in cui gli sviluppatori possono verificare il codice in produzione è quello di nascondere la funzione dietro un flag o un flipper. Il rischio potenziale qui è che il codice può essere abbandonato dietro i flag, risultando in un grande debito tecnico per il codice che non viene rimosso perché è nascosto.

2. **Branching per caratteristica**

Per superare alcune delle limitazioni della strategia del ramo singolo, si possono introdurre due tipi aggiuntivi di rami: rami di funzionalità e rami di integrazione. Tecnicamente, non sono tipi diversi

di rami; è solo la convenzione che indica che tipo di lavoro, si sta effettuando sul ramo.

Nella strategia di distribuzione *branch-per-feature*, tutto il nuovo lavoro è fatto in un ramo *feature*, che è il più piccolo possibile per contenere un'intera idea. Questi rami sono tenuti aggiornati con il lavoro fatto da altri sviluppatori tramite un ramo di integrazione. Quando è il momento di rilasciare il software, il build master può scegliere selettivamente quali feature sono da includere nella compilazione, e creare un nuovo ramo di integrazione per la distribuzione. In Fig. 4.6 vediamo un esempio di questa strategia:

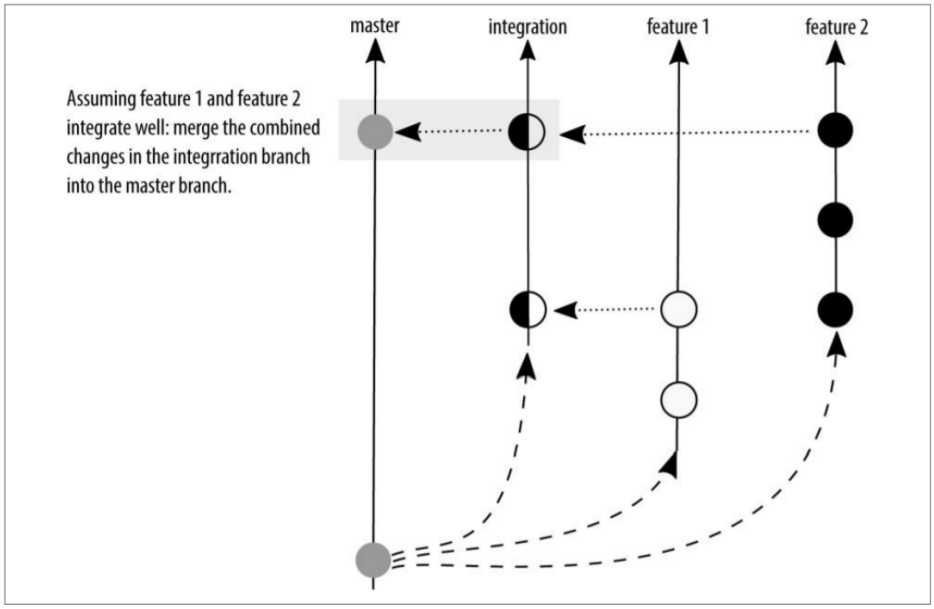

Figura 4.6: Branch-per-feature: i rami delle caratteristiche sono tenuti aggiornati tramite un ramo di integrazione

Aggiungendo rami di funzionalità e un ramo di integrazione, si può continuare ad avere codice pronto per il deployment, ma anche una pausa prima di distribuire il codice. In questo modello, una build viene fatta selezionando quali caratteristiche sono pronte per essere incorporate, il feature branch viene distribuito e se non ci sono errori. Viene unito al master come mostrato nella Fig.4.7. Questo significa

che se ci sono problemi con un ramo feature, il master può essere immediatamente distribuito nuovamente perché è provato che è in uno stato funzionante.

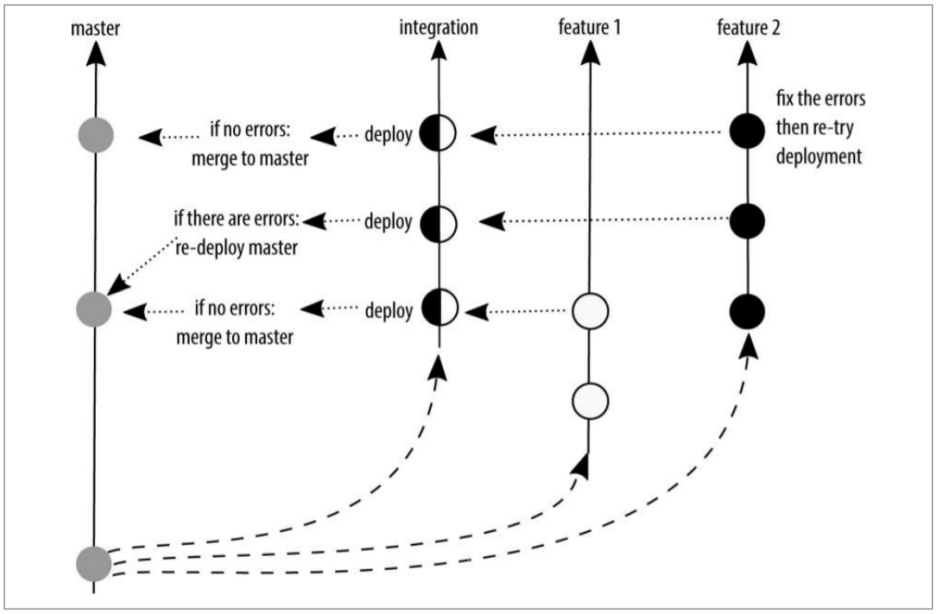

Figura 4.7: I rami delle caratteristiche sono distribuiti dopo una revisione e poi fusi nel master

Vediamo i vantaggi:

- Molto simile allo sviluppo mainline, l'attenzione è sul rapido dispiegamento del codice.
- A differenza dello sviluppo mainline, c'è un passo di compilazione opzionale. Quando il passo di compilazione è usato, c'è la possibilità di selezionare quali caratteristiche dovrebbero essere incorporate nel ramo master per il deployment.

Vediamo gli svantaggi:

- Se il codice viene mantenuto su un ramo delle caratteristiche, ma non viene immediatamente inserito nel master, c'è un requisito di manutenzione extra per gli sviluppatori che hanno bisogno di

mantenere le loro caratteristiche aggiornate in attesa di essere inserite nel ramo distribuito.

- La denominazione semantica dei rami aiuta coloro che hanno familiarità con il sistema, ma rappresenta anche un linguaggio interno che può rendere più difficile l'onboarding se ci sono molte funzioni aperte.
- Ora c'è un requisito di housekeeping per gli sviluppatori di rimuovere i vecchi rami quando vengono inseriti nel master. Questo non è un grosso peso, ma è più di quanto sarebbe richiesto lavorando da un singolo ramo master.

3. **State branching**

   Lo state branching introduce l'idea di un luogo per alcuni dei rami. La Fig.4.8 mostra che il codice è unito da un ramo all'altro, e ognuno dei rami è distribuito in un ambiente specifico

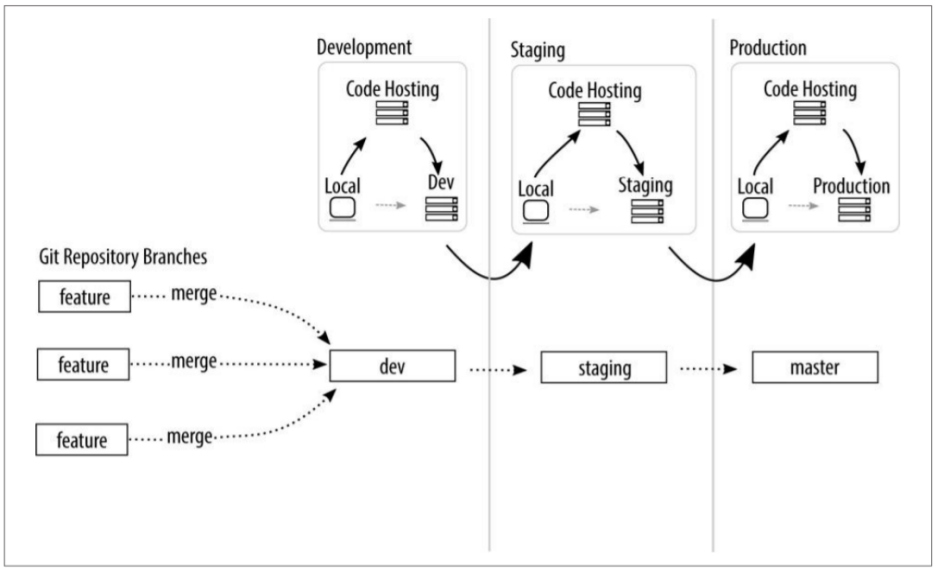

Figura 4.8: Esempio di un vero processo di deployment che usa un sistema centralizzato di hosting del codice

Attraverso le convenzioni di denominazione dei rami, si rende chiaro quale codice sta per essere usato in quale ambiente, e quindi quali

condizioni potrebbero aver bisogno di essere soddisfatte prima di unire i commit. Vediamo la convenzione di nomi usate per branching adattata dal progetto git. Usa quattro rami denominati come segue:

Maint Questo ramo contiene codice dal più recente rilascio stabile di Git, così come ulteriori commit per rilasci puntuali (manutenzione).

Master Questo ramo contiene i commit che dovrebbero andare nella prossima release

Next Questo ramo è destinato a testare argomenti che sono considerevoli per la stabilità nel ramo principale.

Pu Questo è il ramo degli aggiornamenti proposti(Proposted Update), contente commit che non sono ancora pronti per l'inclusione.

I rami funzionano come una piramide impilata. Ognuno dei rami "inferiori" contiene commit che non sono presenti nei rami "superiori", come mostrato in figura

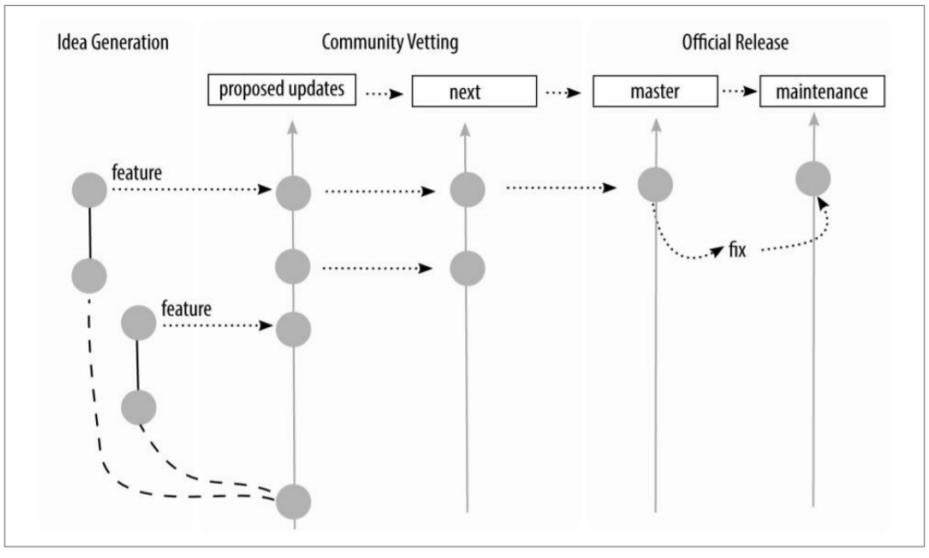

Figura 4.9: Rami di integrazione usati dal progetto Git

Vediamo i vantaggi:

- I nomi dei rami sono specifici del contesto e completamente rilevanti per il lavoro da svolgere
- Poiché i nomi dei rami sono estremamente specifici per il contesto di quel team, può essere più difficile ottenere coerenza tra i progetti, rendendo l'onboarding più difficile.

Vediamo gli svantaggi:

- Non è sempre ovvio da dove iniziare un ramo senza una guida
- Non ci sono congetture sullo scopo di ogni ramo, rendendo più facile per le persone selezionare il ramo giusto quando uniscono il loro lavoro.

4. **Distribuzione programmata**

   La ramificazione dell'implementazione programmata è la strategia più appropriata da usare se non si ha una suite di test completamente automatizzata, e in ogni situazione in cui si deve programmare un'implementazione. Non appena si coinvolgono degli esseri umani in un processo di revisione sul tuo processo di distribuzione, ci saranno inevitabilmente dei ritardi, e si avrà bisogno di un modo per sospendere il lavoro mentre ed aspettare queste persone.

   Vediamo questa convenzione di branching tramite la strategia di GitFlow. All'inizio il progetto software ha un solo ramo, develop. Da questo ramo, i programmatori creano un ramo divergente e aggiungono le loro caratteristiche. Idealmente quando si lavora con un team, una features (in questo caso ha un senso ampio, un refactoring, un correzione di un bug ecc.) sarà descritta in un ticket prima di iniziare il lavoro, e il nome del ramo assomiglierà al nome del ticket, per esempio, se si avesse un ticket "1234" che fosse una segnalazione di bug per correggere un link, e si stesse usando la con-

venzione `[ticketid]-[terse_title]` , il nome del ramo sarebbe `1234-fixing_links]`.

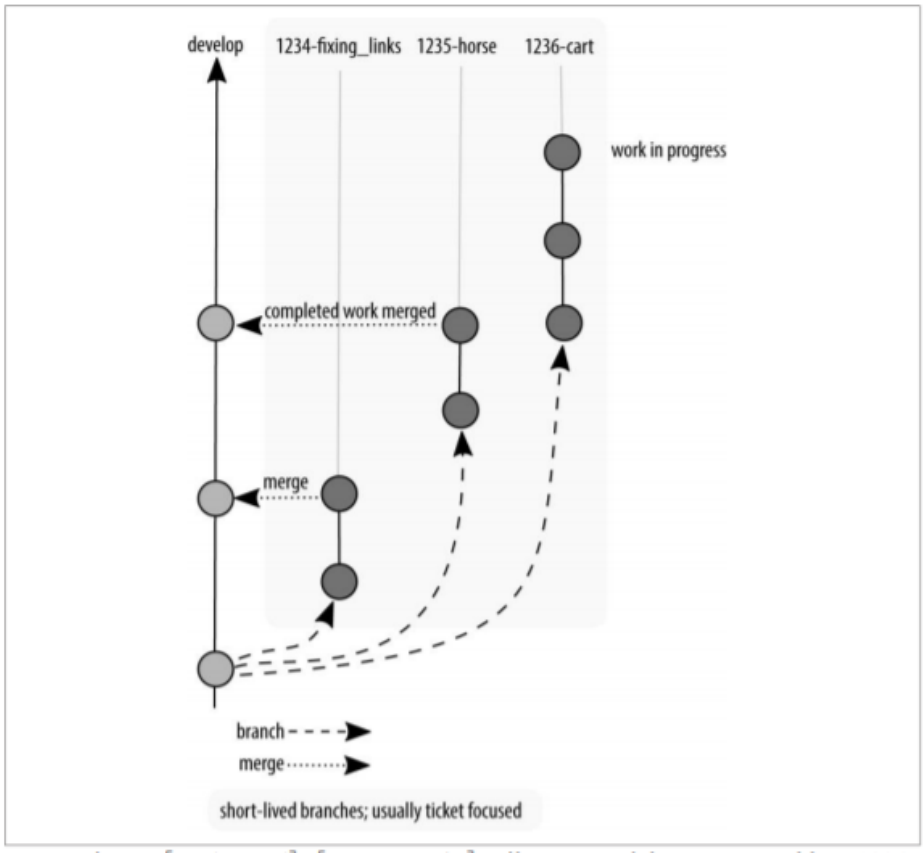

Figura 4.10: Rami di sviluppo e di funzionalità usati in GitFlow

Quando viene raggiunto lo stato di *feature freeze*, si crea un nuovo ramo dove è permesso solo fare correzione di bug, vediamo ora il grafico aggiornato:

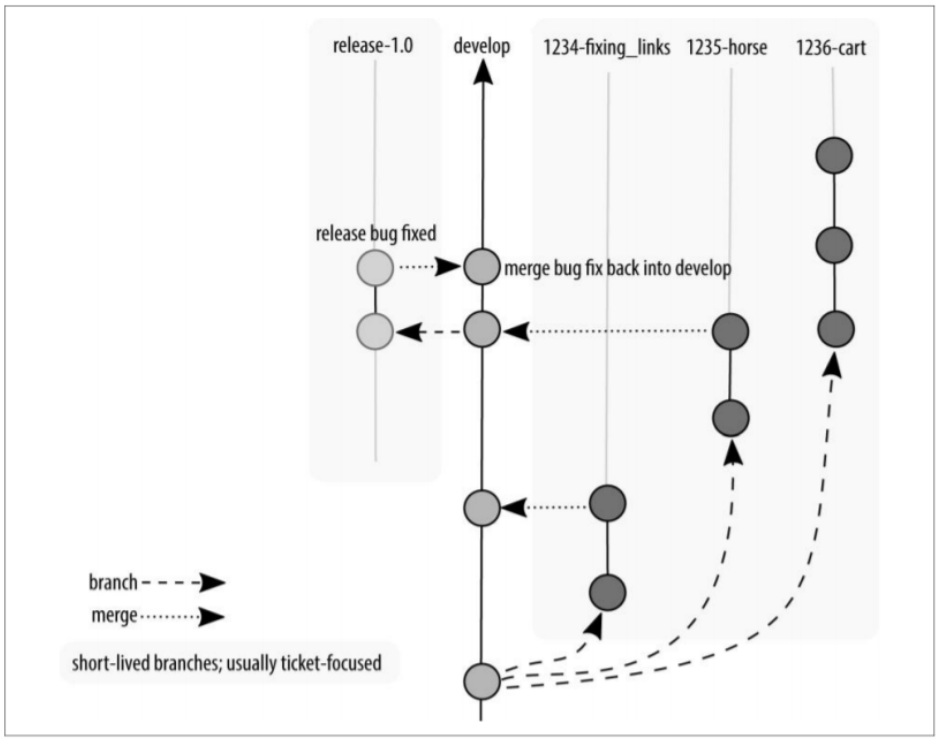

Figura 4.11: Feature freeze in GitFlow; solo le correzioni di bug sono permesse

Magari non tutte le feauture sono state completate quando si è raggiunto il feature freeze. Il lavoro continua nel ramo di sviluppo, eventuali correzioni di bug vengono messe sia nel ramo di distribuzione che nel ramo di sviluppo. Vediamo ora il grafico:

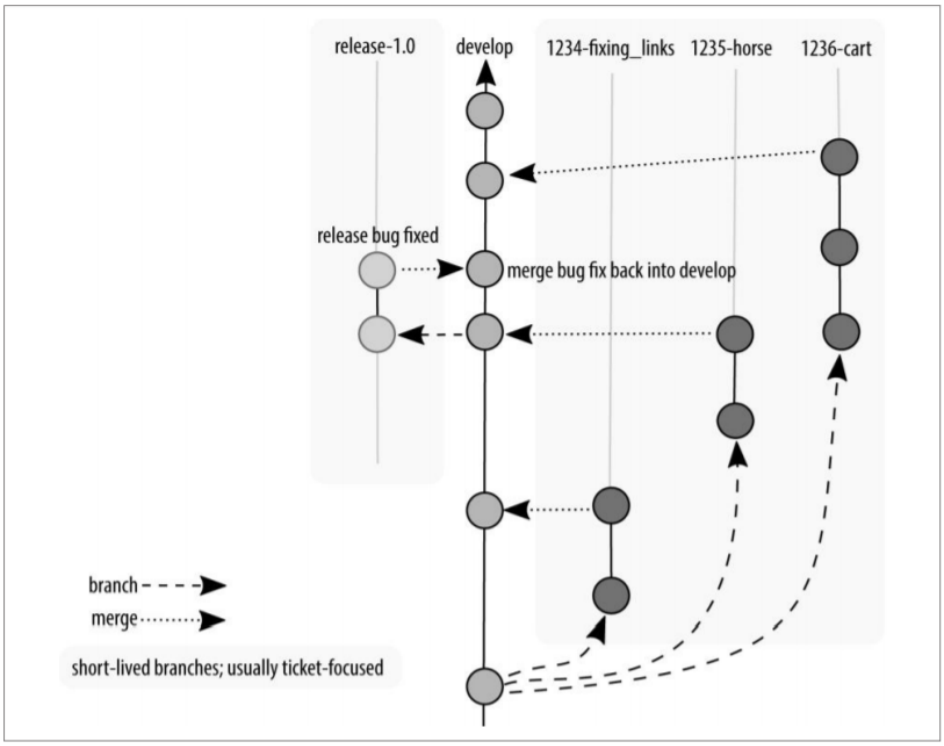

Figura 4.12: Lo sviluppo continua, ma non è incorporato nel ramo di rilascio

Dopo un certo tempo di test, verrà dichiarato che tutti i bug sono stati trovati, e ciò che rimane è pronto per essere distribuito. A questo punto, tutto il codice che ha superato i test di garanzia della qualità viene impegnato in un nuovo ramo, master, che viene poi etichettato con la versione del software in quel punto.

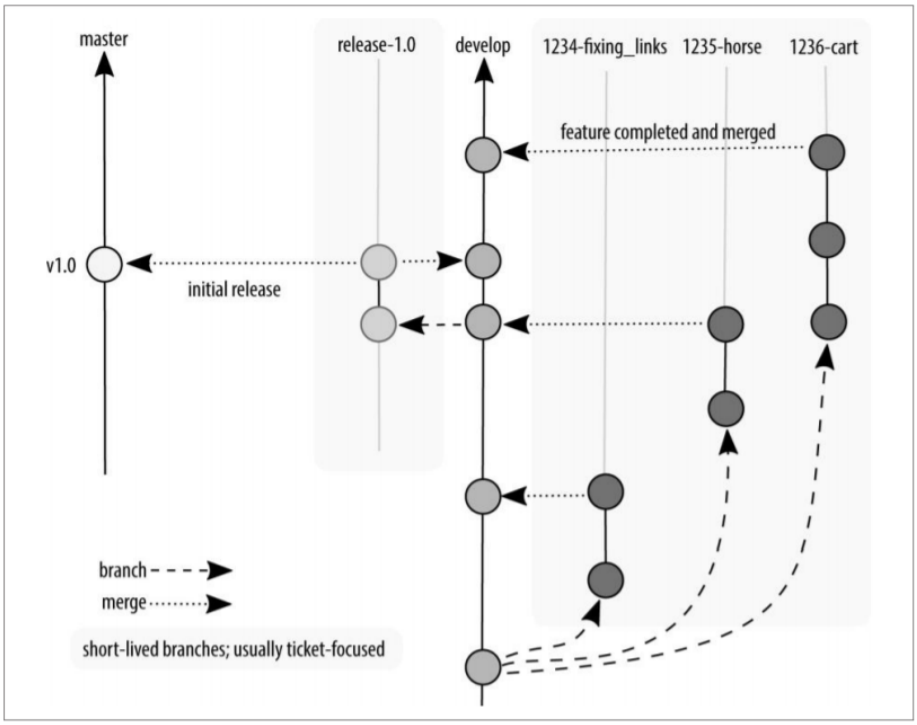

Figura 4.13: Il software viene rilasciato tramite fusione in un nuovo ramo, il master, con un tag

Ora manca l'ultimo pezzo di questa tecnica, dopo il rilascio c'è bisogno di fare degli hotfix, un hotfix è la correzione di un bug così importante che va corretto **immediatamente**. Vediamo il grafico dopo un hotfix:

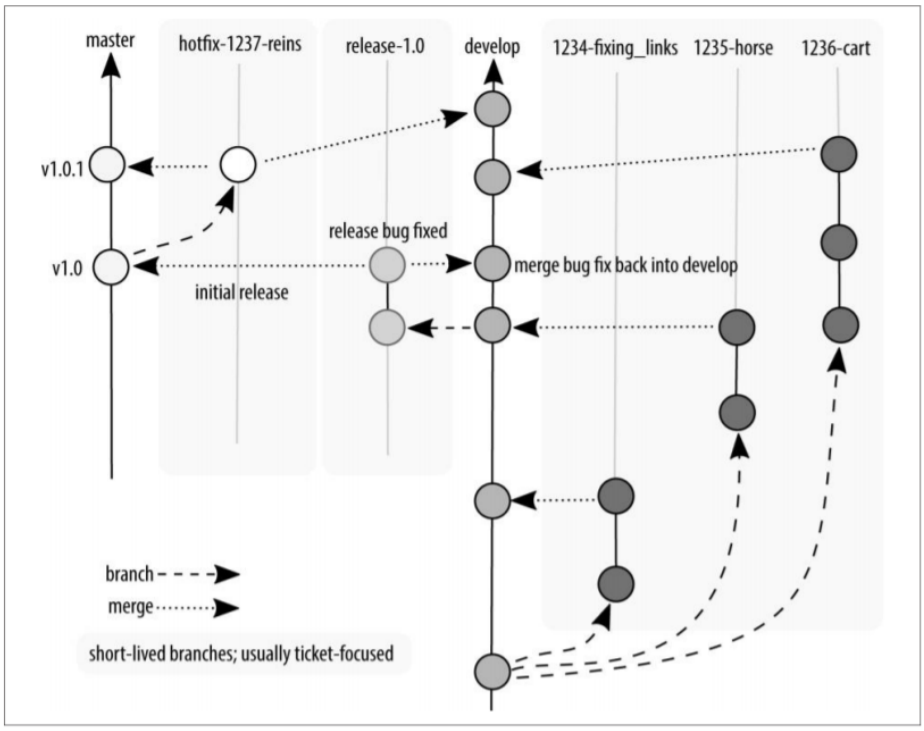

Figura 4.14: Un hotfix è stato fatto, inserito nella master, e il tag di rilascio è ora 1.0.1

Non è necessario creare tutti questi rami all'inizio. In effetti, è meglio se non si fa, perché finisce per essere più codice da mantenere.

Vediamo i vantaggi:

- La distribuzione programmata non richiede un'infrastruttura di test estesa per iniziare ad usarla.
- Il processo di costruzione del software, con fasi di sviluppo, controllo qualità e produzione, è molto comune. Questo significa che le convenzioni di GitFlow risulteranno molto familiari agli sviluppatori di software una volta che avranno capito il processo di come e dove avvengono i loro tipici compiti nella convenzione di ramificazione.
- Aderendo alle convenzioni, gli sviluppatori dovrebbero essere sempre in grado di determinare da quale ramo dovrebbero iniziare il loro lavoro.

- Questo è anche un buon modello per il software in versione, come un prodotto che si scarica da un app store dove non è opportuno distribuire una nuova versione ogni pochi giorni.

Vediamo gli svantaggi:

- C'è un sacco di sovraccarico cognitivo per gli sviluppatori che sono nuovi alla distribuzione del software e non hanno sperimentato il processo di accompagnare un prodotto attraverso ogni fase di sviluppo.
- Se gli sviluppatori iniziano il loro lavoro dal ramo sbagliato, può essere complicato rimettere tutto in sincronia.
- Non è così trendy come il deployment continuo.

4.2 Le carte

In questo capitolo vedremo le carte GitLab, sono delle carte create da me che hanno lo scopo, di spiegare ed aiutare i team nell'utilizzo di GitLab. Queste carte non state approvate ad Essence, sono un primo approccio per unire le carte Essence e gli strumenti (di sviluppo e non), in questo caso GitLab.

4.2.1 Gli alfa

Questa è la prima carta, come possiamo vedere è una carta alfa, e segue la costruzione proposta da Essence. Ha un titolo, il simbolo di stato alfa, una breve descrizione e l'elenco dei possibili stati alfa. Notiamo anche che il colore della carta è il giallo, questo vuol dire che l'area di interessa della carta è: la soluzione. Per effettuare questa scelta, come prima cosa ho fatto un ragionamento ad esclusione. Fra le tre aree di interesse, ho subito escluso quella dei clienti (verde), perché il

cliente non vede mai GitLab (solitamente) e non gli riguarda. Ora la scelta rimane fra, soluzione (giallo) o progetto (blu), ho escluso l'ambito progetto, perché guardando gli alfa blu, ho notato che si riferivano a concetti legati al team e all'ambiente di lavoro. Osservando invece gli alfa gialli ho visto che facevano riferimento a questioni concrete riguardanti lo sviluppo software, quindi ho scelto come area di interesse: soluzione.

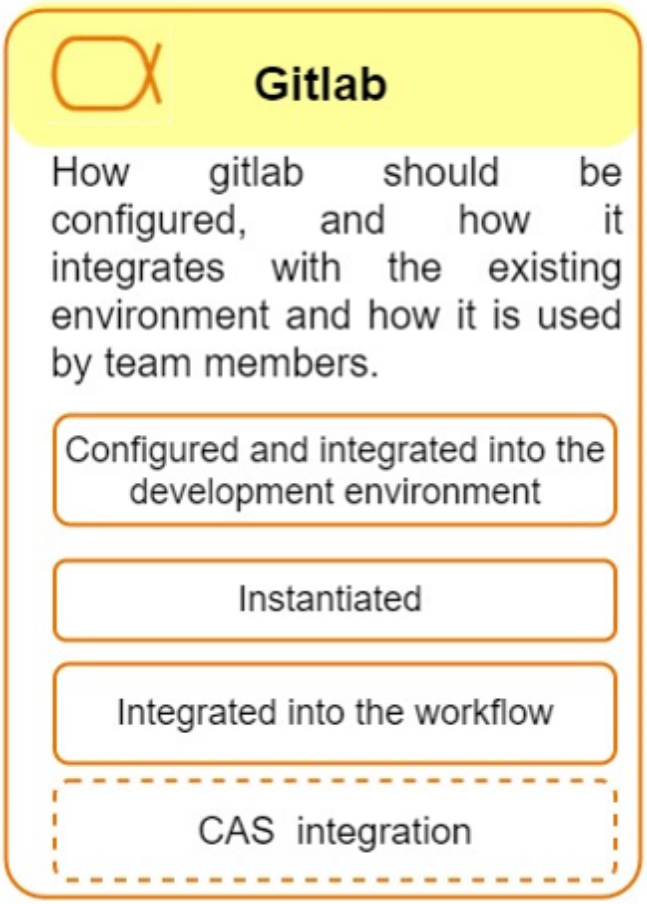

Figura 4.15: L'alfa di GitLab

Ora andrò a spiegare tutte e tre gli stati alfa ed i loro relativi checkpoint. [1]

Vediamo il primo stato alfa: **Configurato e integrato nell'ambiente di sviluppo**

[1]Le carte di questo capitolo sono una creazione dell'autore di questa tesi, non sono state approvate dall'associazione Essence ne da Ivar Jacobson. Sono liberamente usabili

Questo stato è riferito a chi vuole installare GitLab, nel proprio ambiente e non utilizzare GitLab in cloud. Se hai scelto di affidarti ad un server GitLab remoto già messo in piedi da qualcuno, considera questa carta soddisfatta,

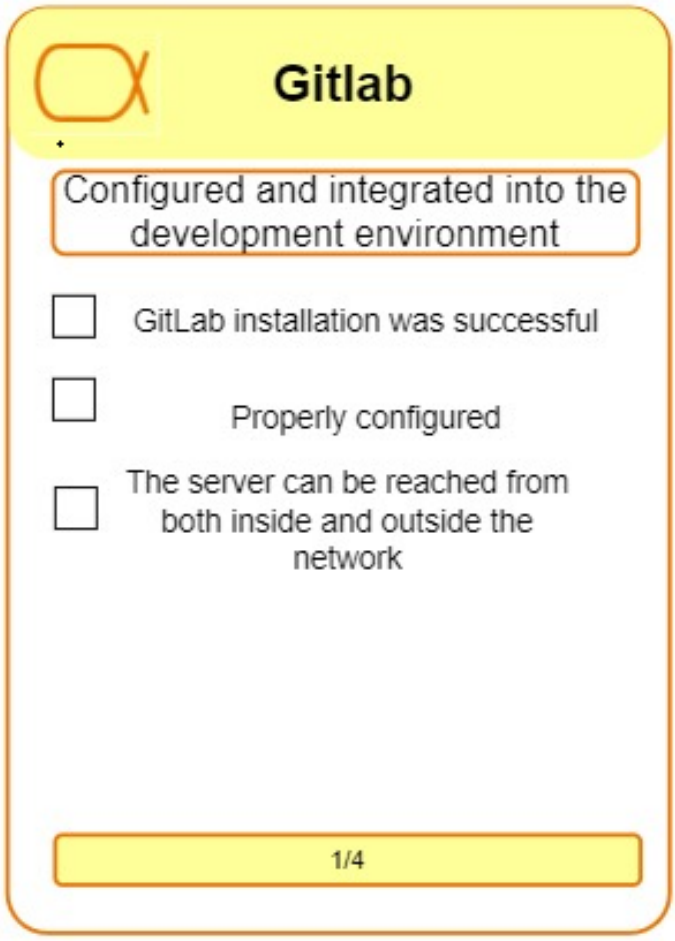

Figura 4.16: Primo stato alfa di GitLab

Il primo checkpoint è considerato soddisfatto se l'installazione di GitLab è avvenuta con successo, questo non vuol dire senza errori, vuol dire che alla fine dell'installazione sarà possibile utilizzare GitLab, eventuali errori non compromettono il corretto utilizzo di GitLab.

Il secondo checkpoint è considerato soddisfatto se dopo l'installazione si è portato a termine anche la configurazione, questo può voler dire tante cose, che dipendono dalle esigenze del team. Ne possiamo riconoscere al-

cune, la creazione degli account, quanti account creare, che privilegi dare ad ogni account, la compilazione dei dettagli per ogni account, ed altro. Si deve anche configurare dall'account amministrativo la dashboard.

L'ultimo checkpoint dice che è possibile raggiungere il server GitLab sia dall'interno che dall'esterno della rete, sempre riferito al fatto che lo stiamo installando su un server nostro. Questo punto ci dice che gli sviluppatori del progetto devono essere in grado di utilizzare GitLab sia all'interno dell'azienda ma anche dall'esterno. Ovviamente questo si adatta all'esigenza del team (e dell'azienda), quindi si può anche scegliere di rendere il server accessibile solo dall'interno o solo dall'esterno, in tutti e due i casi il punto è considerato completato.

Ora andiamo al secondo stato alfa, che si intitola: **Inizializzato**

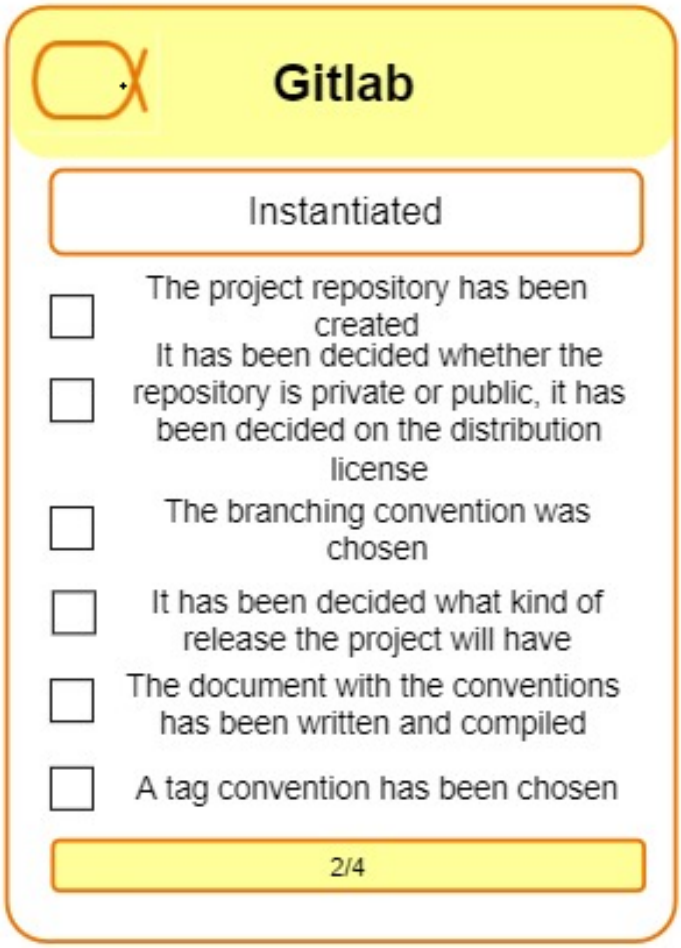

Figura 4.17: Secondo stato alfa di GitLab

In questo stato GitLab è pronto per essere utilizzato ma prima è consigliato completare anche questo stato. In quanto completando tutte i punti si avrà un migliore utilizzo di GitLab durante il progetto.

Il primo punto è abbastanza ovvio, anche se fondamentale.

Il secondo punto consiglia di decidere se il progetto potrebbe essere un progetto consultabile da chiunque (quindi pubblico) o solo dai membri del team (privato quindi). Nulla vieta di cambiare questa decisione in seguito, o di decidere che alcune parti siano pubbliche ed altre private. La cosa importante è cercare di decidere il più possibile prima di iniziare il progetto. Sempre nello stesso punto troviamo la decisione della licenza di distribuzione, c'è ne sono varie tutte diverse fra di loro, per una visione più approfondita leggere il capitolo 4.1.1.

Il terzo punto fa riferimento alla scelta della strategia di branching, ne abbiamo parlato nel capitolo 4.1.2

Il quarto punto invita a decidere che tipo di sviluppo avrà il progetto. Ce ne sono di tanti tipi, sempre in produzione, rilasciare una nuova versione ad intervalli di tempo, esempio ogni mese. Decidere fino a che versione continuare a dare assitenza ed altro. Solitamente è una cosa che decide il project manager oppure il team.

Il quinto punto dice che il documento delle convenzioni è stato scritto e compilato, se non hai presente cos'è un documento delle convenzioni eccoti un esempio:

```
Product Manager: Name
Dev site: URL
Branch deployed on dev site: name of branch
Live site: URL
Branch deployed on live site: name of branch
When starting a dev ticket, branch from: name of branch
When starting a hotfix ticket, branch from: name of branch
When updating your work, use: git command
When merging your work post review, use: git command
```

Figura 4.18: Esempio di un documento delle convenzioni

Questo è solo un esempio, ogni team può aggiungere e togliere degli campi, in base alla propria esperienza ed esigenza.

L'ultimo punto dice che deve essere scelta una convenzione di tag. I tag sono usati per individuare specifici commit. Si può pensare a loro come a un segnalibro. I tag possono essere aggiunti solo a specifici commit. Per sapere a quale commit vuoi aggiungere il tuo tag, probabilmente vorrai usare una combinazione di log e show. Si aggiunge un tag al commit, per catturare particolare punti importanti durante la storia del progetto. Una convenzione è decidere come si chiamano e su quali commit mettere i tag. Decidila insieme al team se non ne avete già una.

Ora vediamo il terzo stato alfa:

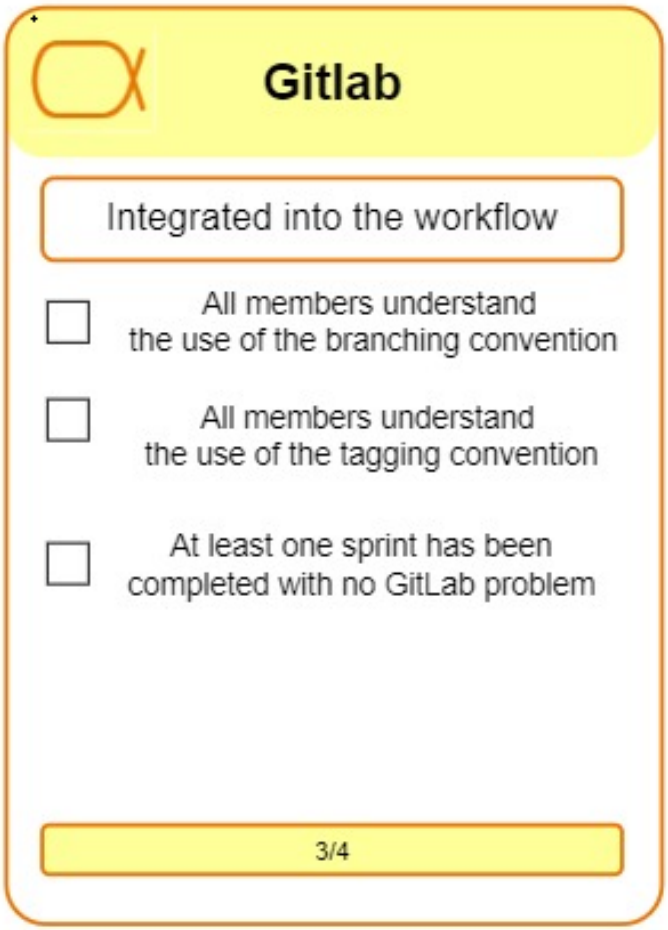

Figura 4.19: Terzo stato alfa di GitLab

Come primo campo abbiamo: tutti i membri del team hanno capito la convenzione di branching. Potremmo considerare raggiunto questo punto, quando non ci sono più errori di rami creati col nome sbagliato, quando non ci sono errori di codice messo nel ramo sbagliato, quando tutti i membri del team non si fanno più domanda, sui rami del progetto.

Il punto due, assomiglia al punto uno. Anche in questo punto potremmo considerarlo soddisfatto se, tutti i membri del team, hanno rispettato e compreso la convenzione quando hanno creato i tag. Se nessuno ha domanda su come creare un tag e quando crearlo.

L'ultimo punto dice: É stato completato almeno uno sprint senza problemi riguardante GitLab.

Come prima cosa dico, che non si parla dei comandi Git, se alcuni membri hanno avuto problemi, perché non si ricordavano un comando, o non riuscivo ad inserire le chiavi SSH, va bene uguale. Mi riferisco ai problemi derivanti da un utilizzo di **team**, per fare un esempio: creazione di una branch che non andava creata. Ho messo poi un fattore temporale, cioè almeno un iterazione, da questo ne deriva che dovreste aspettare almeno un iterazione per considerare concluso questo punto, o guardare ad un iterazione già conclusa. Ho fatto questa scelta perché credo che una fatto pratico sia importante, come ultima verifica di una buona comprensione ed utilizzo.

Se il tuo team non lavora con iterazioni di nessun tipo, non importa, prendete un unità di team e fate un controllo, consiglio due settimane.

Notare che non ho messo nessun punto riguardante i comandi Git. Queste carte non hanno lo scopo di misurare o verificare, se ogni membro del team, conosce o meno Git. In queste carte si guida un team ad usare GitLab, non possono essere usate come guida pratica ai comandi.

Vediamo l'ultimo stato alfa, come notiamo la cornice di questo stato è tratteggiata, questo ci indica che è uno stato opzionale, vediamo tutti

i punti:

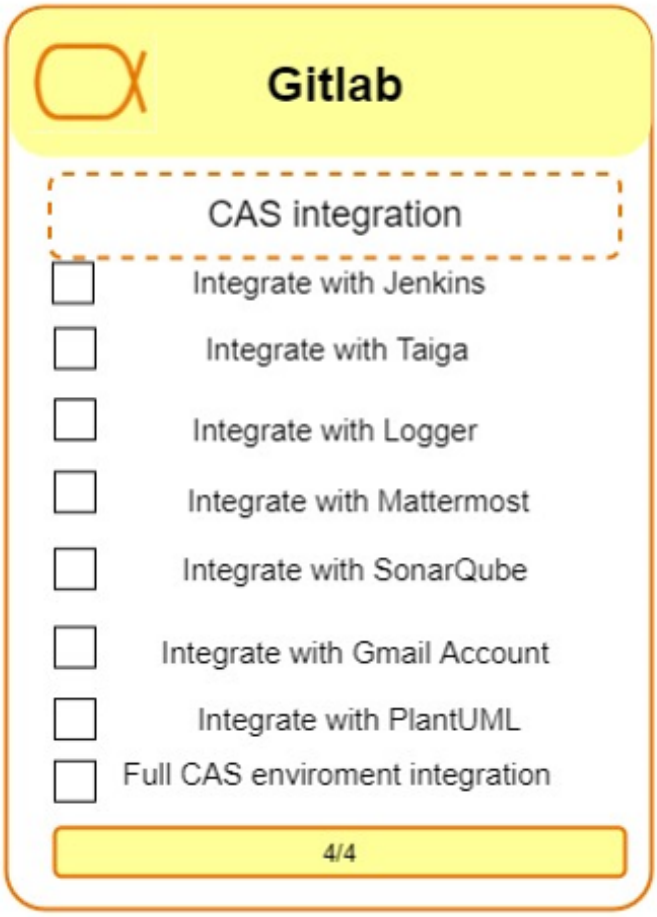

Figura 4.20: Stato opzionale di GitLab

Questo stato serve per far comprendere al team che GitLab può essere integrato con tutto l'ambiente CAS. L'ordine dei punti non è secondario, è in ordine di importanza, dal tool più importante (Jenkins) a quello meno importnate (Plant UML). Vediamo velocemente questi tools per chi non li conoscesse:

- Jenkins

  Jenkins è uno strumento open source di supporto allo sviluppo software scritto in linguaggio Java.

- Taiga

Taiga è un sistema di gestione dei progetti gratuito e open source. Si integra bene con la metodologi agile.

- Logger

  Software open source che raccoglie dati sul codice scritto tramite IDE. Attualmente disponibile per Atom,Eclipse,Intellij.

- Mattermost

  Mattermost è un servizio di chat online open-source auto-gestibile con condivisione di file, ricerca e integrazioni.

- SonarQube

  Software per il controllo continuo della qualità del codice, esegue revisioni automatiche con analisi statica del codice per rilevare bug, smell code e vulnerabilità di sicurezza su oltre 20 linguaggi di programmazione.

- Gmail Account

  Integrazione con l'account di posta elettronica di Google.

- PlantUML

  Software open source che serve principalmente per creare diagrammi UML.

Un punto si considera soddisfatto se l'integrazione è avvenuta con successo, sia da GitLab verso il tool sia dal tool verso GitLab. L'ultimo punto indica che serve fare un controllo che tutti i tool siano integrati fra di loro, ad esempio non solo Jenkins con GitLab ma anche Jenkies con SonarQube. Questo punto si considera soddisfatto se tutti i tool che abbiamo scelto di usare si integrano con GitLab e fra di loro.

Ogni strumento dell'ambiente CAS può essere essenzializzato, nella figura sotto stante possiamo vedere la carta alfa del logger. Data la sua semplicità e il suo utilizzo passivo il logger ha solo uno stato.

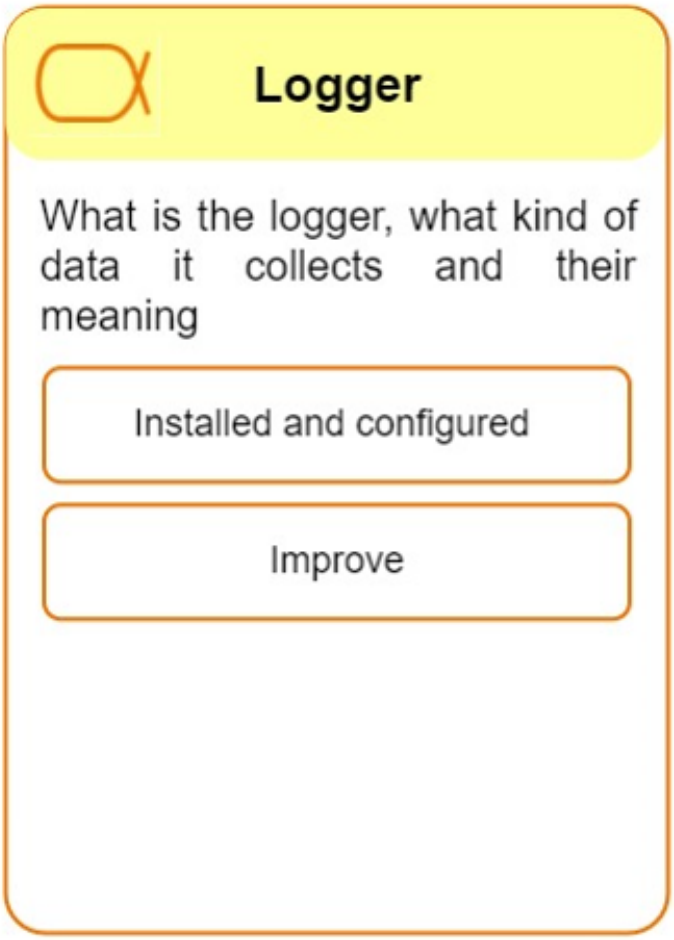

Figura 4.21: Carta Essence del logger

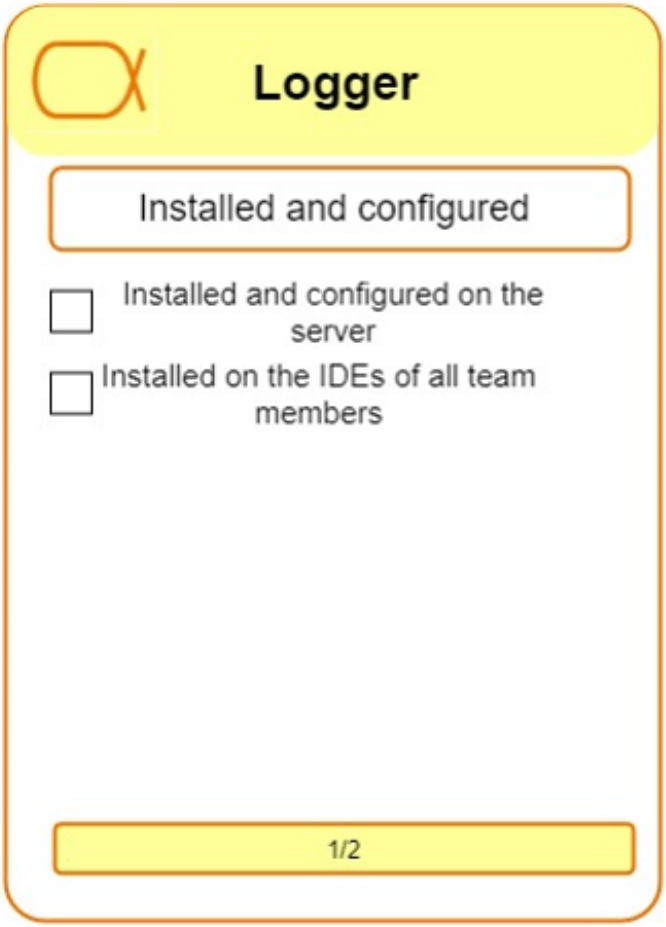

Figura 4.22: Primo stato alfa del logger

Figura 4.23: Secondo stato alfa del logger

4.2.2 Competenza

Vediamo l'altra tipologia di carta, la carta competenza. Nella carta competenza si parla di Git e non di GitLab, ovviamente per usare GitLab è fondamentale saper usare Git.

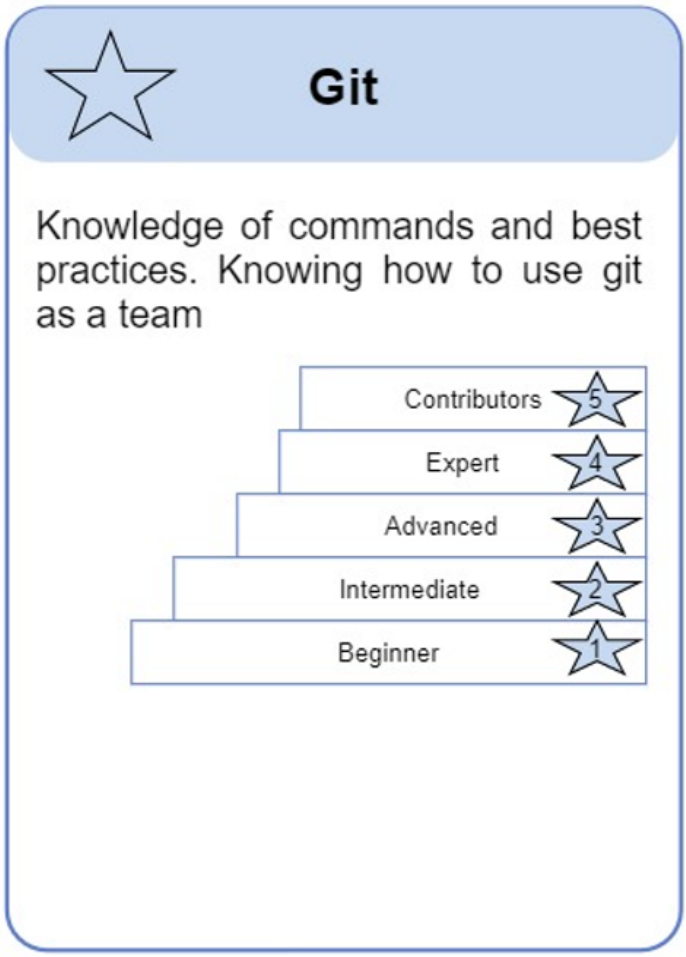

Figura 4.24: Carta competenza di GitLab

Ho identificato cinque livelli di competenza. Vediamo in dettaglio ognuno di essi, e come si può capire in quali di questi livelli ci si trova.

Beginner

Il livello più basso è il principiante. Direi che su questa scala si può considerare principiante chiunque, sappia i comandi base di Git. Creare un repository, fare il clone, creare una branch e sapersi muovere fra di esse, fare il push del proprio codice. Questo è il livello principiante, molto facile da raggiungere. Con questo livello si possono fare dei progetti singolarmente, difficilmente di team.

Intermediate

Il secondo livello è intermedio. In questo livello, ci si aspetta una conoscenza maggiore dei comandi Git, sapere fra un merge o un rabase, sapere fare un fetch, sapere interagire con un server remoto e fare un commit. Con questo livello è possibile creare e partecipare a progetti di gruppo, non troppo complessi.

Advanced

Il terzo livello è avanzato. Chi è in questo livello sa quasi tutto, su Git.Si conoscono tutti i comandi, si ha una discreta esperienza pratica. Si è già lavorato con convenzione di branching e di tag, si sa risolvere un conflitto e come recuperare il lavoro perso per errore. Si conosce la procedura per tornare in uno stato stabile in caso di errori. Si può lavorare su qualsiasi progetto, indipendentemente dalla grandezza

Expert

Il quarto livello è esperto. Ci sono solo due differenze fra esperto ed avanzato.La prima è che l'esperto sa la teoria dietro a Git, l'avanzato so tutti i comandi e quando usarli, però non sa come funziona Git. Non sa alcuni dettagli implementatavi e quindi ignora alcuni processi che succedono in background. La seconda è l'esperienza, sia in termini di progetti, portati a termine, che in termini di anni di lavoro utilizzando Git. Diciamo che ci si può definire esperto, dopo cinque anni di utilizzo di Git.

Contributors

Il quinto ed ultimo livello è il collaboratore. Raggiunge questo livello chi riesci a portare migliorie all'interno del progetto ufficiale Git.

4.3 Un uso delle carte GitLab

Nel progetto di Tweeter tracker, per manutenere e conversare il nostro codice, abbiamo usato GitLab. Ci è stato fornito già pronto all'uso su un server dell'università di Bologna.

4.3.1 Uso e problematiche con GitLab

Come team abbiamo usato GitLab piuttosto male. Come prima cosa abbiamo creato il repository, poi non abbiamo scelto una convenzione di branching, quindi avevamo sei rami, uno per ogni membro del team, più il master. Il ramo di ognuno membro si chiamava col sul nome, questa scelta è stata una pessima scelta, in questo modo ognuno doveva ricordarsi che cosa stavano sviluppando gli altri. Poi se qualcuno passava a uno sviluppo di un altra funzionalità, perché aveva completata quella precedente ci si doveva ricordare di questo cambio. Questo non ha causato molti problemi perché, il codice che caricavamo su GitLab la maggior parte delle volte era stato unito in locale e poi messo nel ramo master. Il merge avveniva nel computer locale di uno di noi, lo testavamo e poi caricavamo il codice nel ramo master. Per passarci il codice, ognuno di noi lo caricava nel proprio ramo e poi la persona incaricata scaricava i vari rami, in modo separato e insieme decidevamo come integrare i vari pezzi di codice fra di loro. Questo perché non sapevamo usare Git molto bene, ed avevamo paura di perdere tempo, o creare disastri con la funzione merge di Git. Dopo la seconda iterazione abbiamo iniziato a dare ai rami i nomi delle featuer, come ad esempio bot o mappa.

Ora vediamo l'opinione personale di un membro del team, riguardante GitLab:

> Avevo già usato GitLab prima del progetto di ingegneria del software. Ce lo avevano spiegato alle superiori. Dopo ho guar-

dato dei tutorial per approfondire. Ho trovato fondamentale GitLab, lo riuserei senza dubbio anche in un progetto futuro. Avrei preferito che avessero spiegato come utilizzarlo, almeno qualche direttiva per un utilizzo più professionale in team. Non ho mai fatto un merge, avevo paura di perdere troppo tempo a correggere conflitti. Anche la paura di sbagliare qualcosa durante il merge mi ha frenato molto. Se dovessi dare un voto da uno a dieci, su come mi sono trovato ad utilizzare GitLab, darei un otto: mi sono trovato molto bene ma sono convinto che se imparassi ad usarlo, in maniera professionale, potrei sfruttare tutte le sue potenzialità.

Vediamone una seconda, di un altro membro:

Sapevo già usare Git ma veramente poco. Ho imparato informandomi su internet e chiedendo ai miei compagni di team, forse - ripensandoci - un po' troppo spesso. Credo che GitLab sia lo strumento perfetto per manutenere il codice, anche se ci ho messo non poco per sentirmi prendere fiducia, credo sia uno degli strumenti essenziali per un programmatore. La cosa che mi ha creato più difficoltà agli inizi è stato imparare i comandi Git, all'università non ce li avevano mai spiegati e alle superiori solo in modo superficiale. In più era passato tanto tempo. I primi tempi ho usato l'interfaccia grafica di GitLab per eseguire quasi tutte le operazioni, principalmente commit. Non ho avuto problemi con GitLab in sé, anche perché molto simile a GitHub. Se dovessi dare un voto darei un sette, non sapendo bene come utilizzarlo ho dovuto perdere del tempo nel capirlo, e questo mi ha creato un po' frustrazione, perché avrei preferito usare quel tempo per fare altro.

Come i miei compagni, do anche io una mia opinione:

Considero GitLab uno strumento fondamentale, che sicuramente vale la pena approfondire. La piattaforma GitLab mi piace molto, non ha creato problemi di nessun tipo, e fornisce delle statistiche molto interessanti riguardanti il progetto. Anche io avrei preferito che ci avessero spiegato (anche non nel corso di ing. del software), come utilizzarlo in modo professionale, almeno una spiegazione di base per quelli che non avevano mai utilizzato Git. Se dovessi dare un voto, darei un sette, se avessi avuto una conoscenza migliore dei comandi Git, sicuramente il voto sarebbe stato più alto.

4.3.2 Uso delle carte in una retrospettiva

In questo paragrafo vedremo come le carte GitLab avrebbero potuto aiutare nel progetto. Faremo una simulazione di una retrospettiva, aggiungendo le carte GitLab.

Simulazione di una retrospettiva

Simuliamo la prima retrospettiva. Ricordo che nel primo sprint non avevamo scritto neanche una riga di codice. Oltre ai sette alfa, aggiungo anche l'alfa di GitLab, visto nel paragrafo precedente. Ora giochiamo a progressive Poker, con il nuovo alfa, come abbiamo fatto per tutti gli altri. Ogni giocatore si chiede in che stato si è attualmente, molto probabilmente tutti concorderebbero sul fatto che si è nel secondo stato (ricordo che essere in uno stato, significa avere completato tutti i punti di quello precedente, ma non tutti quelli dello stato attuale).

Il primo stato lo consideriamo completato, in quanto l'ambiente GitLab era già installato e configurato nell'ambiente CAS, i team lo usavano soltanto. Ora il team che concorda sullo stato, legge i vari punti da soddisfare per completare anche quello stato, leggendo i vari punti ha un idea

di cosa andrà fatto, per usare al meglio GitLab. Esaminiamo i primi due punti. Il primo è banale e non dici niente al team che probabilmente non sapesse già, il secondo è più interessante ma è il progetto esigeva una progetto privato con l'invito dei due professori, e la licenza non è importante in quanto il progetto non vedrà mai la luce del sole. In sintesi i primi due punti il team li considera soddisfatti.

Proseguiamo, valutando i punti tre e quattro. Il terzo dice che è stata scelta una convenzione di branching. Ora il team non sa cos'è una convenzione di branching, né tanto me ne conosce una, fa una veloce ricerca su Google e scopre che in realtà ne conosceva una a grandi linee, che è la *branch per feature*. Nessuno la conosce in dettaglio, quindi decidono di adottarla e poi lasciare il compito di uno studio più approfondito a un membro del team, che successivamente la spiegherà al resto del team. In questo modo il team comprende cos'è una convenzione di branching, come si usa e la sua utilità. La scelta risolve il problema dei nomi poco significativi, e di dove mettere quale codice, che invece il team ha riscontrato nel vero sviluppo.

Ora il punto quattro, che è abbastanza facile da considerare concluso. Non necessita di una discussione, in quanto il tempo per ogni sprint e la consegna del progetto è stata deciso dal docente/PO.

Il quinto punto necessita invece di una ricerca, in quanto il team non sa cosa sia un documento delle convenzioni. Anche in questo caso viene scelto un membro del team che approfondisce e successivamente riporta al resto del gruppo (magari non viene scelta la stessa persona del punto tre).

La scrittura di un documento di convenzione, e la sua successiva condivisone, ad esempio mettendolo nel master, nella prima pagina, faciliterà la lettura del codice a chi non fa parte del team, ad esempio i docenti.

L'ultimo punto vine affrontato come i punti tre e e cinque. Il team

che non conosce nessuna convenzione di tag, decide di lasciare che un membro del team approfondisca e poi spieghi a tutti.

In conclusione il team, ritiene di aver completato i punti uno, due, quattro del secondo stato alfa.

La Fig.4.25 mostra il risultato della prima retrospettiva inclusa la carta GitLab:

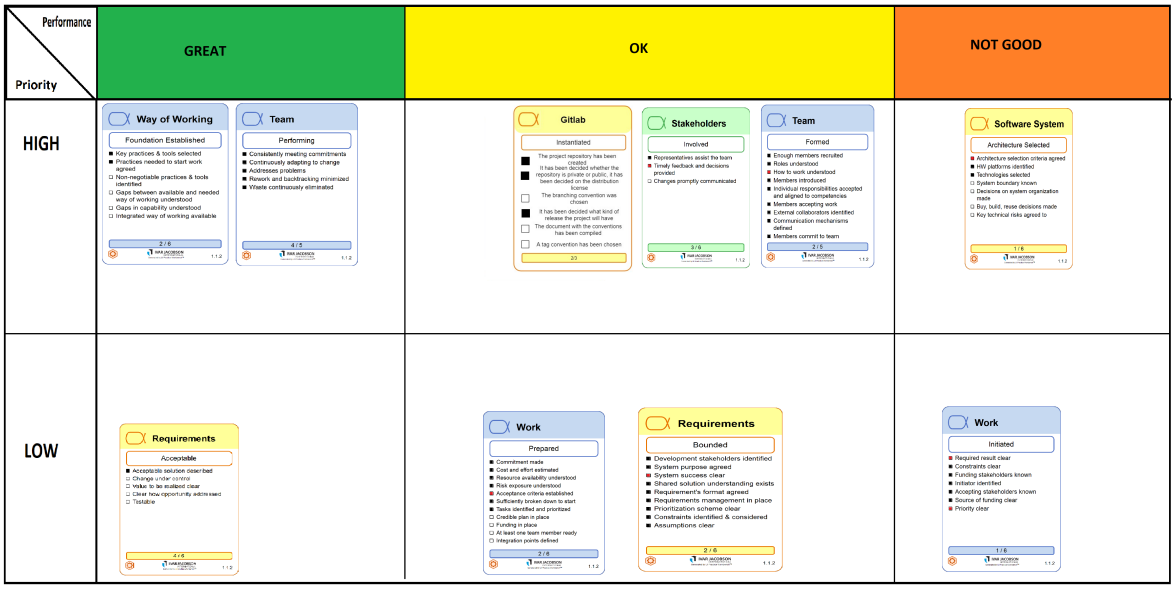

Figura 4.25: Prima retrospettiva che include la carta GitLab

In conclusione, credo che le carte GitLab costituiscano un prezioso ausilio verso un uso più consapevole dello strumento. Aiuteranno i team meno esperti nel capire quali passaggi fare per sfruttare al meglio tutte le potenzialità che offre questo strumento complesso. La semplicità che si ottiene usando le carte, piuttosto che cercare su internet, o improvvisando, migliora la velocità con cui un team lavora, e crea meno frustrazione all'interno del team.

4.4 Conclusioni

In questo capitolo finale, ho mostro e spiegato le carte GitLab create da me. Prima di tutto spiego i concetto di GitLab utili per un lavoro di gruppo, le licenze di distribuzione, il modello di collaborazione e le strategie di branching. Proseguo mostrando le carte e spiegandole una ad una, faccio vedere anche le carte Essence di un altro strumento dell'ambiente CAS, il logger. Questo serve oltre per un utilizzo pratico che ogni strumento può essere essensializzato. Concludo facendo una simulazione di come il nostro gruppo avrebbe potuto usare le carte GitLab nel progetto, tramite una retrospettiva, mostrando gli eventuali vantaggi.

Capitolo 5

Conclusioni & Ringraziamenti

Di questa tesi vorrei che rimanesse un'immagine del futuro dell'ingegneria del software. Essence sta cercando di fondare una teoria condivisa in questo oceano di confusione e di idee.

Un altro punto che vorrei che rimanesse è il potenziale di Essence, sembrano solo un gioco di carte, ma fornisce un potente strumento per apprendere nuove pratiche, per gestire al meglio i progetti facendo in modo di renderli più efficienti, riducendo il numero di progetti che falliscono.

Ma sono sempre delle carte, quindi ci si può giocare e divertire. Rendendo il lavoro un po' meno faticoso e piacevole e a tratti divertente. In un mondo dove le parole divertimento e lavoro sono agli antipodi, io credo fermamente che se ci si diverte mentre si lavora la propria vita non potrà che essere felice. Credo che Essence possa rendere questo meraviglioso mondo dello sviluppo software meno faticoso, più divertente e giocoso, senza rinunciare alla serietà di cui gode e merita.

Ringrazio i proff. Ciancarini e Missiroli, per avermi guidato in questa esperienza. Ringrazio i miei amici Gabriele, Francesca, Nicholas che hanno reso questo percorso ricco di risate e divertimento, ed mi hanno aiutato molto durante lo studio: grazie! non ce l'avrei fatta senza di voi.

Ringrazio la mia famiglia che mi ha dato questa fantastica opportunità e ha creduto in me.

Bibliografia

[1] Ivar Jacobson. Essence Language Key. https://practicelibrary.ivarjacobson.com/content/essence-language-key.

[2] Ivar Jacobson. Games of Essence. https://essence.ivarjacobson.com/alphastatecards.

[3] Ivar Jacobson, Harold Lawson, Panwei Ng, Paul McMahon, and Michael Goedicke. *The Essentials of Modern Software Engineering*. ACM Books. Morgan & Claypool Publishers, 2019.

[4] Ivar Jacobson, Pan-Wei Ng, Paul E McMahon, Ian Spence, and Svante Lidman. The Essence of software engineering: the SEMAT kernel. *Communications of the ACM*, 55(12):42–49, 2012.

[5] OMG. Essence – Kernel and Language for Software Engineering Methods 1.2. https://www.omg.org/spec/Essence/, 2018.

[6] P. Ciancarini and M. Missiroli. Teaching the Essence of Software Development. In *Proc. 32nd Conf on Software Engineering Education and Training CSEE&T*, pages 1–2. IEEE, 2020.

[7] Brian Kerrian Spence. Practice patience game. https://essence.ivarjacobson.com/publications/blog/bette -scrum-through-essence-part-2.

[8] Shirley Gregor. The nature of theory in information systems. *MIS Quarterly*, pages 611–642, 2006.

[9] Ivar Jacobson Pontus Johnson, Mathias Ekstedt. Where's the Theory for Software Engineering? *IEEE Software*, 29(5):96–96, 2012.

[10] Edsger W Dijkstra. Go-to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[11] Emma Jane Hogbin Westby. *Git for Teams*. O'Reilly Media, Inc., 2015.

[12] Inc. GitHub and You! How to choose a license. https://choosealicense.com/.