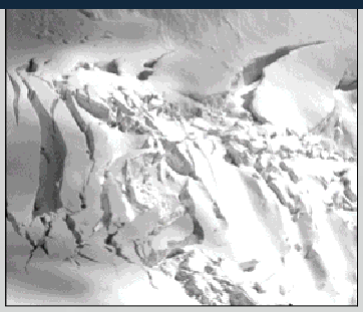
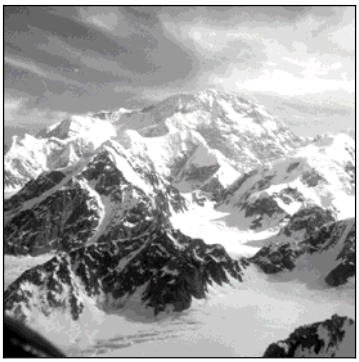


PARTE II
Affrontare la complessità

4

4



Requisiti Elicitazione

Un errore comune che la gente commette quando cerca di progettare qualcosa di completamente infallibile è quello di sottovalutare l'ingegnosità dei completi idioti.

-Douglas Adams, in *Mostly Harmless*

Arequirement è una caratteristica che il sistema deve avere o un vincolo che deve soddisfare per essere accettato dal cliente. L'**ingegneria** dei **requisiti** mira a definire i requisiti del sistema in costruzione. L'ingegneria dei requisiti comprende due attività principali; *Elicitazione dei requisiti*, che si traduce nella specificazione del sistema che il cliente comprende, e *analisi*, che si traduce in un modello di analisi che gli sviluppatori possono interpretare senza ambiguità. L'elicitazione dei requisiti è la più impegnativa delle due, perché richiede la collaborazione di diversi gruppi di partecipanti con background diversi. Da un lato, il cliente e gli utenti sono esperti nel loro settore e hanno un'idea generale di ciò che il sistema dovrebbe fare, ma spesso hanno poca esperienza nello sviluppo di software. Dall'altro lato, gli sviluppatori hanno esperienza nella costruzione di sistemi, ma spesso hanno poca conoscenza dell'ambiente quotidiano degli utenti.

Scenari e casi d'uso forniscono strumenti per colmare questa lacuna. Uno *scenario* descrive un esempio di utilizzo del sistema in termini di una serie di interazioni tra l'utente e il sistema. Un *caso d'uso* è un'astrazione che descrive una classe di scenari. Sia gli scenari che i casi d'uso sono scritti in linguaggio naturale, una forma comprensibile per l'utente.

In questo capitolo ci concentriamo sull'evocazione dei requisiti basati su scenari. Gli sviluppatori sollevano i requisiti osservando e intervistando gli utenti. Gli sviluppatori rappresentano prima di tutto i processi di lavoro attuali dell'utente come scenari così come sono, poi sviluppano scenari visionari che descrivono le funzionalità che il sistema futuro dovrà fornire. Il cliente e gli utenti convalidano la descrizione del sistema rivedendo gli scenari e testando piccoli prototipi forniti dagli sviluppatori. Man mano che la definizione del sistema matura e si stabilizza, gli sviluppatori e il cliente concordano una specifica dei requisiti sotto forma di requisiti funzionali, requisiti non funzionali, casi d'uso e scenari.

4.1 Introduzione: Esempi di usabilità

Piedi o miglia? [a]

Durante un esperimento laser, un raggio laser è stato diretto verso uno specchio sullo Space Shuttle Discovery. Il test prevedeva che il raggio laser venisse riflesso verso la cima di una montagna. L'utente ha inserito l'elevazione della montagna come "10.023", supponendo che le unità di misura dell'ingresso fossero in piedi. Il computer ha interpretato il numero in miglia e il raggio laser è stato riflesso lontano dalla Terra, verso un'ipotetica montagna alta 10.023 miglia.

Punto decimale contro il separatore delle migliaia

Negli Stati Uniti, i punti decimali sono rappresentati da un punto (".") e migliaia di separatori sono rappresentati da una virgola (","). In Germania, il punto decimale è rappresentato da una virgola e i mille separatori da un punto. Si supponga che un utente in Germania, consapevole di entrambe le convenzioni, stia visualizzando un catalogo online con i prezzi indicati in dollari. Quale convenzione dovrebbe essere usata per evitare confusione?

Modelli standard

Nell'editor di testo Emacs, il comando <Control-x><Control-c> esce dal programma. Se è necessario salvare dei file, l'editor chiede all'utente: "Salva file myDocument.txt? (y o n)". Se l'utente risponde y, l'editor salva il file prima di uscire. Molti utenti si basano su questo schema e digitano sistematicamente la sequenza <Control-x><Control-c> seguita da una "y" quando si esce da un editor. Altri editor, invece, chiedono quando si esce dalla domanda: "Sei sicuro di voler uscire? (y o n)". Quando gli utenti passano da Emacs a un tale editor, non riusciranno a salvare il loro lavoro finché non riusciranno a rompere questo schema.

a. Esempi da [Nielsen, 1993] e [Neumann, 1995].

L'elicitazione dei requisiti riguarda la comunicazione tra sviluppatori, clienti e utenti per definire un nuovo sistema. La mancata comunicazione e la mancata comprensione dei domini altrui si traduce in un sistema difficile da usare o che semplicemente non supporta il lavoro dell'utente. Gli errori introdotti durante l'elicitazione dei requisiti sono costosi da correggere, poiché di solito vengono scoperti in ritardo nel processo, spesso tanto quanto la consegna. Tali errori includono funzionalità mancanti che il sistema avrebbe dovuto supportare, funzionalità che sono state specificate in modo errato, interfacce utente fuorvianti o inutilizzabili e funzionalità obsolete. I metodi di selezione dei requisiti mirano a migliorare la comunicazione tra sviluppatori, clienti e utenti. Gli sviluppatori costruiscono un modello del dominio applicativo osservando gli utenti nel loro ambiente. Gli sviluppatori selezionano una rappresentazione che sia comprensibile per i clienti e gli utenti (ad esempio, scenari e casi d'uso). Gli sviluppatori convalidano il modello del dominio applicativo costruendo semplici prototipi dell'interfaccia utente e raccogliendo il feedback dei potenziali utenti. Un esempio di un semplice prototipo è il layout di un'interfaccia utente con voci di menu e pulsanti. Il potenziale utente può manipolare le voci di menu e i pulsanti per avere un'idea dell'uso del sistema, ma non c'è una risposta effettiva dopo che i pulsanti sono stati cliccati, perché la funzionalità richiesta non è implementata.

La sezione 4.2 fornisce una panoramica dei requisiti richiesti e del loro rapporto con le altre attività di sviluppo. La sezione 4.3 definisce i concetti utilizzati in questo capitolo. La sezione 4.4 discute le attività di sollecitazione dei requisiti. La sezione 4.5 discute le attività di

gestione relative all'individuazione dei requisiti. La sezione 4.6 tratta il caso di studio di ARENA.

4.2 Una panoramica dei requisiti Elicitazione

L'**elicitazione dei requisiti si** concentra sulla descrizione dello scopo del sistema. Il cliente, gli sviluppatori e gli utenti identificano un'area problematica e definiscono un sistema che affronta il problema. Tale definizione è chiamata **specifica dei requisiti** e serve come contratto tra il cliente e gli sviluppatori. La specifica dei requisiti è strutturata e formalizzata durante l'analisi (Capitolo 5, *Analisi*) per produrre un **modello di analisi** (vedi Figura 4-1). Sia la specifica dei requisiti che il modello di analisi rappresentano le stesse informazioni. Essi differiscono solo nel linguaggio e nella notazione che usano; la specifica dei requisiti è scritta in linguaggio naturale, mentre il modello di analisi è solitamente espresso in una notazione formale o semiformale. La specifica dei requisiti supporta la comunicazione con il cliente e gli utenti. Il modello di analisi supporta la comunicazione tra gli sviluppatori. Sono entrambi modelli del sistema nel senso che cercano di rappresentare accuratamente gli aspetti esterni del sistema. Dato che entrambi i modelli rappresentano gli stessi aspetti del sistema, l'elicitazione dei requisiti e l'analisi dei requisiti si verificano contemporaneamente e in modo iterativo.

L'analisi e la selezione dei requisiti si concentrano solo sulla visione del sistema da parte dell'utente. Per esempio, la funzionalità del sistema, l'interazione tra l'utente e il sistema, gli errori che il sistema può rilevare e gestire, e le condizioni ambientali in cui le funzioni del sistema fanno parte dei requisiti. La struttura del sistema, la tecnologia di implementazione scelta per costruire il sistema, la progettazione del sistema, la metodologia di sviluppo e altri aspetti non direttamente visibili all'utente non fanno parte dei requisiti.

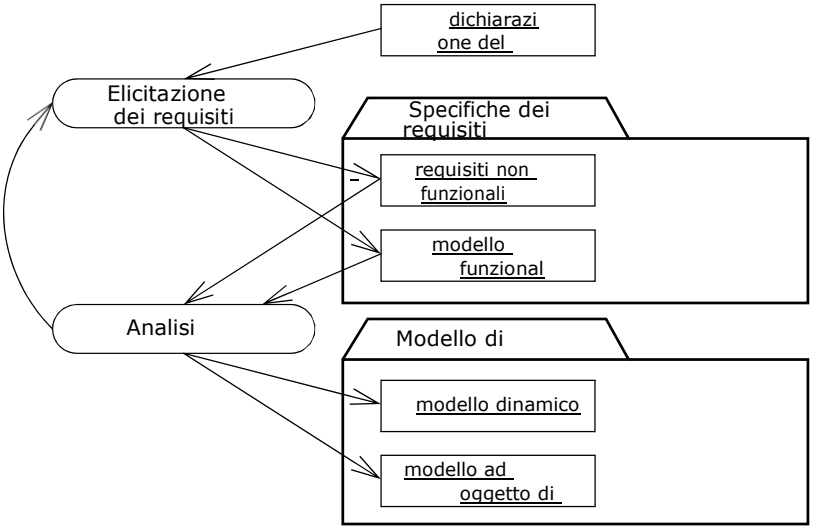


Figura 4-1 Prodotti dell'elicitazione e dell'analisi dei requisiti (diagramma di attività UML).

L'elicitazione dei requisiti comprende le seguenti attività:

- *Identificare gli attori*. Durante questa attività, gli sviluppatori identificano i diversi tipi di utenti che il futuro sistema supporterà.
- *Identificare gli scenari*. Durante questa attività, gli sviluppatori osservano gli utenti e sviluppano una serie di scenari dettagliati per le funzionalità tipiche fornite dal sistema futuro. Gli scenari sono esempi concreti del futuro sistema in uso. Gli sviluppatori utilizzano questi scenari per comunicare con l'utente e approfondire la loro comprensione del dominio applicativo.
- *Identificare i casi d'uso*. Una volta che gli sviluppatori e gli utenti si mettono d'accordo su un insieme di scenari, gli sviluppatori derivano dagli scenari un insieme di casi d'uso che rappresentano completamente il sistema futuro. Mentre gli scenari sono esempi concreti che illustrano un singolo caso, i casi d'uso sono astrazioni che descrivono tutti i casi possibili. Quando descrivono i casi d'uso, gli sviluppatori determinano la portata del sistema.
- *Perfezionare i casi d'uso*. Durante questa attività, gli sviluppatori si assicurano che le specifiche dei requisiti siano complete, dettagliando ogni caso d'uso e descrivendo il comportamento del sistema in presenza di errori e condizioni eccezionali.
- *Identificare le relazioni tra i casi d'uso*. Durante questa attività, gli sviluppatori identificano le dipendenze tra i casi d'uso. Essi consolidano anche il modello dei casi d'uso, calcolando le funzionalità comuni. Questo assicura che le specifiche dei requisiti siano coerenti.
- *Identificare i requisiti non funzionali*. Durante questa attività, gli sviluppatori, gli utenti e i clienti concordano su aspetti visibili all'utente, ma non direttamente legati alla funzionalità. Questi includono i vincoli sulle prestazioni del sistema, la sua documentazione, le risorse che consuma, la sua sicurezza e la sua qualità.

Durante la selezione dei requisiti, gli sviluppatori hanno accesso a molte fonti di informazione diverse, compresi i documenti forniti dai clienti sul dominio applicativo, i manuali e la documentazione tecnica dei sistemi legacy che il sistema futuro sostituirà, e soprattutto, gli utenti e i clienti stessi. Gli sviluppatori interagiscono maggiormente con gli utenti e i clienti durante la selezione dei requisiti. Ci concentriamo su due metodi per ottenere informazioni, prendere decisioni con utenti e clienti e gestire le dipendenze tra i requisiti e altri artefatti:

- **Joint Application Design (JAD) si** concentra sulla creazione di un consenso tra sviluppatori, utenti e clienti sviluppando congiuntamente le specifiche dei requisiti. [1]
- La **tracciabilità si** concentra sulla registrazione, la strutturazione, il collegamento, il raggruppamento e il mantenimento delle dipendenze tra i requisiti e tra i requisiti e gli altri prodotti di lavoro.

1. Si noti che l'uso del termine "design" in JAD è un termine improprio: non ha nulla a che vedere con il nostro

uso del termine nei capitoli successivi sulla progettazione di sistemi e oggetti.

4.3 Requisiti Concetti di Elicitazione

In questa sezione, descriviamo i principali concetti di elicitazione dei requisiti utilizzati in questo capitolo. In particolare, descriviamo

- Requisiti funzionali (paragrafo 4.3.1)
- Requisiti non funzionali (Sezione 4.3.2)
- Completezza, coerenza, chiarezza e correttezza (Sezione 4.3.3)
- Realismo, verificabilità e tracciabilità (Sezione 4.3.4)
- Ingegneria greenfield, reingegnerizzazione e ingegneria delle interfacce (Sezione 4.3.5).

Descriviamo le attività di sollecitazione dei requisiti nella Sezione 4.4.

4.3.1 Requisiti funzionali

I requisiti funzionali descrivono le interazioni tra il sistema e il suo ambiente indipendentemente dalla sua implementazione. L'ambiente include l'utente e qualsiasi altro sistema esterno con cui il sistema interagisce. Per esempio, la Figura 4-2 è un esempio di requisiti funzionali per SatWatch, un orologio che si resetta da solo senza l'intervento dell'utente:

SatWatch è un orologio da polso che visualizza l'ora in base alla sua posizione attuale. SatWatch utilizza il GPS
satelliti (Global Positioning System) per determinare la sua posizione e le strutture dati interne per convertire questa posizione in un fuso orario.

Le informazioni memorizzate in SatWatch e la sua precisione di misurazione del tempo sono tali che il proprietario dell'orologio non ha mai bisogno di reimpostare l'ora. SatWatch regola l'ora e la data visualizzate quando il proprietario dell'orologio attraversa i fusi orari e i confini politici. Per questo motivo, SatWatch non ha pulsanti o controlli a disposizione dell'utente.

SatWatch determina la sua posizione utilizzando i satelliti GPS e, in quanto tale, soffre delle stesse limitazioni di tutti gli altri dispositivi GPS (ad esempio, impossibilità di determinare la posizione in determinate ore del giorno in regioni montuose). Durante i periodi di blackout, SatWatch presuppone che non attraversi un fuso orario o un confine politico. SatWatch corregge il proprio fuso orario non appena termina un periodo di blackout.

SatWatch ha un display a due righe che mostra, sulla riga superiore, l'ora (ora, minuti, secondi, fuso orario) e sulla riga inferiore, la data (giorno, data, mese, anno). La tecnologia di visualizzazione utilizzata è tale che il proprietario dell'orologio può vedere l'ora e la data anche in condizioni di scarsa luminosità.

Quando i confini politici cambiano, il proprietario dell'orologio può aggiornare il software dell'orologio utilizzando il dispositivo WebifyWatch (fornito con l'orologio) e un personal computer collegato a Internet.

Figura 4-2 Requisiti funzionali per SatWatch.

I requisiti funzionali di cui sopra si concentrano solo sulle possibili interazioni tra SatWatch e il suo mondo esterno (cioè il proprietario dell'orologio, il GPS e WebifyWatch). La

descrizione di cui sopra non si concentra su nessuno dei dettagli di implementazione (ad esempio, processore, lingua, tecnologia di visualizzazione).

4.3.2 Requisiti non funzionali

I requisiti non funzionali descrivono aspetti del sistema che non sono direttamente correlati al comportamento funzionale del sistema. I requisiti non funzionali includono un'ampia varietà di requisiti che si applicano a molti aspetti diversi del sistema, dall'usabilità alle prestazioni. Il modello FURPS+2 utilizzato dal processo unificato [Jacobson et al., 1999] fornisce le seguenti categorie di requisiti non funzionali:

- L'**usabilità** è la facilità con cui un utente può imparare ad operare, preparare gli ingressi e interpretare le uscite di un sistema o di un componente. I requisiti di usabilità includono, ad esempio, le convenzioni adottate dall'interfaccia utente, la portata della guida in linea e il livello di documentazione per l'utente. Spesso i clienti affrontano i problemi di usabilità richiedendo allo sviluppatore di seguire le linee guida dell'interfaccia utente su schemi di colori, loghi e font.
- L'**affidabilità** è la capacità di un sistema o di un componente di svolgere le funzioni richieste in condizioni dichiarate per un determinato periodo di tempo. I requisiti di affidabilità comprendono, ad esempio, un tempo medio accettabile per un guasto e la capacità di rilevare guasti specifici o di resistere a specifici attacchi alla sicurezza. Più di recente, questa categoria è spesso sostituita dall'**affidabilità**, che è la proprietà di un sistema informatico tale che si può legittimamente fare affidamento sul servizio che fornisce. L'affidabilità comprende l'affidabilità, la **robustezza** (il grado in cui un sistema o un componente può funzionare correttamente in presenza di input non validi o di condizioni ambientali stressanti) e **la sicurezza** (una misura dell'assenza di conseguenze catastrofiche per l'ambiente).
- I requisiti di **prestazione** riguardano gli attributi quantificabili del sistema, come il **tempo di risposta** (la velocità con cui il sistema reagisce all'input dell'utente), la **produttività (quanto** lavoro il sistema può svolgere in un determinato lasso di tempo), la **disponibilità** (il grado in cui un sistema o un componente è operativo e accessibile quando richiesto per l'uso) e **l'accuratezza**.
- I requisiti di **supporto** riguardano la facilità delle modifiche al sistema dopo l'implementazione, tra cui, ad esempio, l'**adattabilità** (la capacità di cambiare il sistema per affrontare ulteriori concetti di dominio applicativo), la **manutenibilità (la capacità di cambiare il sistema per** affrontare nuove tecnologie o correggere difetti), e l'internazionalizzazione (la capacità di cambiare il sistema per affrontare ulteriori convenzioni internazionali, come lingue, unità e formati numerici). La norma ISO 9126 sulla qualità del software [ISO Std. 9126], simile al modello FURPS+, sostituisce questa categoria con due categorie: **manutenibilità** e **portabilità** (la facilità con cui un sistema o un componente può essere trasferito da un ambiente hardware o software ad un altro).

2. FURPS+ è un acronimo che utilizza la prima lettera delle categorie di requisiti: Funzionalità, Usabilità, Affidabilità, Prestazioni e Sostenibilità. Il + indica le sottocategorie aggiuntive. Il modello FURPS è stato originariamente proposto da [Grady, 1992]. Le definizioni in questa sezione sono citate da [IEEE Std. 610.12-1990].

Il modello FURPS+ fornisce ulteriori categorie di requisiti tipicamente incluse anche sotto l'etichetta generale di requisiti non funzionali:

- **I requisiti di implementazione** sono vincoli all'implementazione del sistema, compreso l'uso di strumenti specifici, linguaggi di programmazione o piattaforme hardware.
- **I requisiti di interfaccia** sono vincoli imposti da sistemi esterni, compresi i sistemi legacy e i formati di interscambio.
- **I requisiti operativi** sono vincoli per l'amministrazione e la gestione del sistema nel contesto operativo.
- **I requisiti di imballaggio** sono vincoli alla consegna effettiva del sistema (ad es. vincoli sui supporti di installazione per l'impostazione del software).
- **I requisiti legali** riguardano le questioni di licenza, regolamentazione e certificazione. Un esempio di requisito legale è che il software sviluppato per il governo federale degli Stati Uniti deve essere conforme alla Sezione 508 del Rehabilitation Act del 1973, che richiede che i sistemi informativi del governo siano accessibili alle persone con disabilità.

I requisiti non funzionali che rientrano nelle categorie URPS sono chiamati **requisiti di qualità del** sistema. I requisiti non funzionali che rientrano nelle categorie di implementazione, interfaccia, operazioni, imballaggio e legali sono chiamati **vincoli** o **pseudo requisiti**. I requisiti di budget e di programmazione di solito non sono trattati come requisiti non funzionali, in quanto vincolano gli attributi dei progetti (vedi Capitolo 14, *Project Management*). La Figura 4-3 illustra i requisiti non funzionali per SatWatch.

Requisiti di qualità per SatWatch

- Qualsiasi utente che sappia leggere un orologio digitale e capisca le abbreviazioni internazionali dei fusi orari dovrebbe poter utilizzare SatWatch senza il manuale d'uso. [Requisiti di usabilità]
- Poiché il SatWatch non ha pulsanti, non dovrebbe verificarsi alcun guasto del software che richieda il reset dell'orologio. [Requisito di affidabilità]
- SatWatch dovrebbe visualizzare il fuso orario corretto entro 5 minuti dalla fine di un periodo di blackout del GPS. [Requisito di prestazione]
- SatWatch dovrebbe misurare il tempo entro 1/100 di secondo in 5 anni. [Requisito di prestazione]
- SatWatch dovrebbe visualizzare correttamente l'ora in tutti i 24 fusi orari. [Requisito di prestazione]
- SatWatch dovrebbe accettare gli aggiornamenti a bordo tramite l'interfaccia seriale Webify Watch. [Requisito di sopportabilità]

Vincoli per SatWatch

- Tutti i software correlati associati a SatWatch, incluso il software di bordo, saranno scritti utilizzando Java, in conformità con la politica aziendale in vigore. [Requisito di implementazione]
- SatWatch è conforme alle interfacce fisiche, elettriche e software definite da WebifyWatch API 2.0. [Requisito dell'interfaccia]

Figura 4-3 Requisiti non funzionali per SatWatch.

4.3.3 Completezza, coerenza, chiarezza e correttezza

I requisiti sono continuamente convalidati con il cliente e l'utente. La validazione è una fase critica del processo di sviluppo, dato che sia il cliente che lo sviluppatore dipendono dalle specifiche dei requisiti. La validazione dei requisiti implica la verifica che la specifica sia completa, coerente, inequivocabile e corretta. È **completa** se vengono descritti tutti i possibili scenari attraverso il sistema, compreso il comportamento eccezionale (cioè, tutti gli aspetti del sistema sono rappresentati nel modello dei requisiti). La specifica dei requisiti è **coerente** se non è in contraddizione con se stessa. La specifica dei requisiti è **univoca** se viene definito esattamente un sistema (cioè non è possibile interpretare la specifica in due o più modi diversi). Una specifica è **corretta** se rappresenta accuratamente il sistema di cui il cliente ha bisogno e che gli sviluppatori intendono costruire (cioè, tutto nel modello dei requisiti rappresenta accuratamente un aspetto del sistema per la soddisfazione sia del cliente che dello sviluppatore). Queste proprietà sono illustrate nella Tabella 4-1.

La correttezza e la completezza di una specifica dei requisiti sono spesso difficili da stabilire, soprattutto prima che il sistema esista. Dato che la specifica dei requisiti serve come base contrattuale tra il cliente e gli sviluppatori, la specifica dei requisiti deve essere

Tabella 4-1 Proprietà delle specifiche controllate durante la validazione.

Tutte le caratteristiche di interesse sono descritte dai requisiti. *Esempio di incompletezza:* La specifica `SatWatch` non specifica il comportamento del confine quando l'utente si trova entro i limiti di precisione GPS del confine di uno stato. *Soluzione:* Aggiungere un requisito funzionale che stabilisca che l'ora rappresentata da `SatWatch` non dovrebbe cambiare più di una volta molto spesso di 5 minuti.
Coerente - Non ci sono due requisiti della specifica che si contraddicono a vicenda. *Esempio di incoerenza:* Un orologio che non contiene alcun difetto del software non deve necessariamente fornire un meccanismo di aggiornamento per scaricare nuove versioni del software. *Soluzione:* Rivedere uno dei requisiti contrastanti del modello (ad esempio, riformulare il requisito relativo al fatto che l'orologio non contenga difetti, in quanto non è comunque verificabile).
Univoco: un requisito non può essere interpretato in due modi che si escludono a vicenda. *Esempio di ambiguità:* La specifica `SatWatch` si riferisce ai fusi orari e ai confini politici. Il `SatWatch` si occupa dell'ora legale o no? *Soluzione:* Chiarire il concetto ambiguo per selezionare uno dei fenomeni che si escludono a vicenda (ad esempio, aggiungere il requisito che `SatWatch` debba occuparsi dell'ora legale).
Corretto: i requisiti descrivono le caratteristiche del sistema e dell'ambiente di interesse per il cliente e lo sviluppatore, ma non descrivono altre caratteristiche non volute. *Esempio di guasto:* Ci sono più di 24 fusi orari. Diversi paesi e territori (ad es. l'India) si trovano mezz'ora prima di un fuso orario vicino.

attentamente esaminato da entrambe le parti. Inoltre, le parti del sistema che presentano un rischio elevato devono essere prototipate o simulate per dimostrare la loro fattibilità o per ottenere un feedback dall'utente. Nel caso del SatWatch descritto sopra, un modello dell'orologio verrebbe costruito utilizzando un orologio tradizionale e gli utenti intervistati per raccogliere le loro prime impressioni. L'utente può osservare che desidera che l'orologio sia in grado di visualizzare sia il formato americano che quello europeo della data.

4.3.4 Realismo, verificabilità e tracciabilità

Tre altre tre proprietà desiderabili di una specifica dei requisiti sono che sia realistica, verificabile e tracciabile. La specifica dei requisiti è **realistica** se il sistema può essere implementato entro i vincoli. La specifica dei requisiti è **verificabile** se, una volta che il sistema è stato costruito, i test ripetibili possono essere progettati per dimostrare che il sistema soddisfa le specifiche dei requisiti. Ad esempio, un tempo medio di guasto di cento anni per SatWatch sarebbe difficile da verificare (supponendo che sia realistico in primo luogo). I seguenti requisiti sono ulteriori esempi di requisiti non verificabili:

- *Il prodotto deve avere una buona interfaccia utente*.
- *Il prodotto deve essere privo di errori.*
- *Il prodotto risponde all'utente con 1 secondo per la maggior parte dei casi*.- "La maggior parte dei casi" non è definita.

Una specifica dei requisiti è **rintracciabile** se ogni requisito può essere rintracciato durante tutto lo sviluppo del software alle sue corrispondenti funzioni di sistema e se ogni funzione di sistema può essere rintracciata al suo corrispondente insieme di requisiti. La tracciabilità include anche la capacità di tracciare le dipendenze tra i requisiti, le funzioni di sistema e gli artefatti di progettazione intermedi, compresi i componenti di sistema, le classi, i metodi e gli attributi degli oggetti. La tracciabilità è fondamentale per lo sviluppo dei test e per la valutazione delle modifiche. Nello sviluppo dei test, la tracciabilità consente al tester di valutare la copertura di un caso di test, cioè di identificare quali requisiti sono testati e quali no. Quando si valutano i cambiamenti, la tracciabilità consente all'analista e agli sviluppatori di identificare tutti i componenti e le funzioni del sistema che il cambiamento avrebbe un impatto.

4.3.5 Ingegneria greenfield, reingegnerizzazione e ingegneria delle interfacce

Le attività di sollecitazione dei requisiti possono essere classificate in tre categorie, a seconda della fonte dei requisiti. Nell'**ingegneria greenfield**, lo sviluppo parte da zero - non esiste un sistema precedente - quindi i requisiti vengono estratti dagli utenti e dal cliente. Un progetto di ingegneria greenfield viene attivato da un bisogno dell'utente o dalla creazione di un nuovo mercato. SatWatch è un progetto di ingegneria greenfield.

Un progetto di **reingegnerizzazione** è la riprogettazione e la reimplementazione di un sistema esistente innescato da abilitatori tecnologici o da processi aziendali [Hammer & Champy, 1993]. A volte, la funzionalità del nuovo sistema viene estesa, ma lo scopo essenziale del

il sistema rimane lo stesso. I requisiti del nuovo sistema sono estratti da un sistema esistente.

Un progetto di **ingegneria dell'interfaccia** è la riprogettazione dell'interfaccia utente di un sistema esistente. Il sistema legacy viene lasciato intatto, ad eccezione della sua interfaccia, che viene riprogettata e reimplementata. Questo tipo di progetto è un progetto di reingegnerizzazione in cui il sistema legacy non può essere scartato senza comportare costi elevati.

Sia nella reingegnerizzazione che nell'ingegneria greenfield, gli sviluppatori devono raccogliere quante più informazioni possibili dal dominio applicativo. Queste informazioni si possono trovare nei manuali di procedura, nella documentazione distribuita ai nuovi dipendenti, nel manuale del sistema precedente, nei glossari, nei fogli di calcolo e nelle note sviluppate dagli utenti e nei colloqui con gli utenti e i clienti. Si noti che, sebbene le interviste con gli utenti siano uno strumento prezioso, non riescono a raccogliere le informazioni necessarie se non vengono poste le domande pertinenti. Gli sviluppatori devono prima acquisire una solida conoscenza del dominio applicativo prima di poter utilizzare l'approccio diretto.

In seguito, descriviamo le attività di sollecitazione dei requisiti.

4.4 Requisiti Attività di Elicitazione

In questa sezione, descriviamo le attività di sollecitazione dei requisiti. Queste tracciano una dichiarazione di un problema (vedi Capitolo 3, *Organizzazione e comunicazione del progetto*) in una specifica dei requisiti che rappresentiamo come un insieme di attori, scenari e casi d'uso (vedi Capitolo 2, *Modellazione con UML*). Discutiamo l'euristica e i metodi per ottenere i requisiti dagli utenti e modellare il sistema in termini di questi concetti. Le attività di sollecitazione dei requisiti includono

- Identificazione degli attori (Sezione 4.4.1)
- Identificare gli scenari (Sezione 4.4.2)
- Identificare i casi d'uso (Sezione 4.4.3)
- Casi d'uso di raffinazione (Sezione 4.4.4)
- Identificare i rapporti tra gli attori e i casi d'uso (Sezione 4.4.5)
- Identificazione degli oggetti di analisi iniziale (Sezione 4.4.6)
- Identificazione dei requisiti non funzionali (Sezione 4.4.7).

I metodi descritti in questa sezione sono adattati da OOSE [Jacobson et al., 1992], dal Processo di sviluppo del software unificato [Jacobson et al., 1999], e dalla progettazione guidata dalla responsabilità [Wirfs-Brock et al., 1990].

4.4.1 Identificazione degli attori

Gli attori rappresentano entità esterne che interagiscono con il sistema. Un attore può essere umano o un sistema esterno. Nell'esempio di `SatWatch`, il proprietario dell'orologio, i satelliti GPS e il dispositivo seriale WebifyWatch sono attori (vedi Figura 4-4). Tutti scambiano informazioni con il `SatWatch`. Si noti, tuttavia, che tutti hanno interazioni specifiche con il `SatWatch`: l'orologio

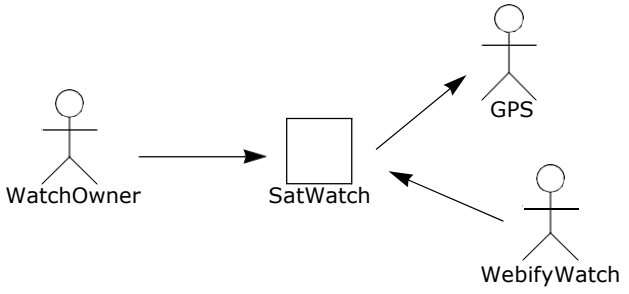


Figura 4-4 Attori per il sistema SatWatch. WatchOwner sposta l'orologio (possibilmente attraverso i fusi orari) e lo consulta per sapere che ora è. SatWatch interagisce con il GPS per calcolare la sua posizione. WebifyWatch aggiorna i dati contenuti nell'orologio per riflettere i cambiamenti nella politica del tempo (ad esempio, cambiamenti nelle date di inizio e fine dell'ora legale).

La proprietaria indossa e guarda il suo orologio; l'orologio monitora il segnale dei satelliti GPS; il WebifyWatch scarica nuovi dati nell'orologio. Gli attori definiscono le classi di funzionalità.

Consideriamo un esempio più complesso, FRIEND, un sistema informativo distribuito per la gestione degli incidenti [Bruegge et al., 1994]. Esso comprende molti attori, come FieldOfficer, che rappresenta la polizia e i vigili del fuoco che rispondono a un incidente, e Dispatcher, l'ufficiale di polizia responsabile di rispondere alle chiamate al 911 e di inviare le risorse per un incidente. FRIEND supporta entrambi gli attori tenendo traccia degli incidenti, delle risorse e dei piani di intervento. Ha anche accesso a diversi database, come un database di materiali pericolosi e procedure per le operazioni di emergenza. Il FieldOfficer e gli attori del Dispatcher interagiscono attraverso diverse interfacce: I FieldOfficer accedono a FRIEND attraverso un assistente personale mobile, i Dispatcher accedono a FRIEND attraverso una postazione di lavoro (vedi Figura 4-5).

Gli attori sono astrazioni di ruoli e non sono necessariamente direttamente collegati alle persone. La stessa persona può ricoprire il ruolo di FieldOfficer o Dispatcher in momenti diversi. Tuttavia, la funzionalità a cui accedono è sostanzialmente diversa. Per questo motivo, questi due ruoli sono modellati come due attori diversi.

Il primo passo dell'elicitazione dei requisiti è l'identificazione degli attori. Questo serve sia per definire i confini del sistema sia per trovare tutte le prospettive da cui gli sviluppatori

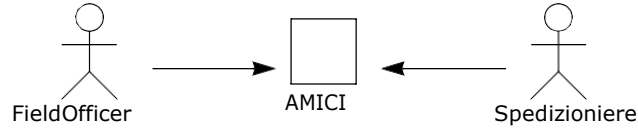


Figura 4-5 Attori del sistema FRIEND. Gli agenti sul campo non solo hanno accesso a diverse funzionalità, ma utilizzano diversi computer per accedere al sistema.

devono prendere in considerazione il sistema. Quando il sistema viene implementato in un'organizzazione esistente (come un'azienda), la maggior parte degli attori di solito esiste prima che il sistema venga sviluppato: essi corrispondono ai ruoli all'interno dell'organizzazione.

Durante le fasi iniziali di identificazione dell'attore, è difficile distinguere gli attori dagli oggetti. Ad esempio, un sottosistema di database può a volte essere un attore, mentre in altri casi può essere parte del sistema. Si noti che una volta definito il confine del sistema, non ci sono problemi a distinguere tra attori e componenti del sistema come oggetti o sottosistemi. Gli attori sono al di fuori del confine del sistema; sono esterni. I sottosistemi e gli oggetti sono all'interno del confine del sistema; sono interni. Così, ogni sistema software esterno che usa il sistema da sviluppare è un attore. Quando si identificano gli attori, gli sviluppatori possono porre le seguenti domande:

Domande per l'identificazione degli attori

- Quali gruppi di utenti sono supportati dal sistema per svolgere il loro lavoro?
- Quali gruppi di utenti eseguono le funzioni principali del sistema?
- Quali gruppi di utenti svolgono funzioni secondarie, come la manutenzione e l'amministrazione?
- Con quale sistema hardware o software esterno il sistema interagirà?

Nell'esempio di FRIEND, queste domande portano a una lunga lista di potenziali attori: vigili del fuoco, agenti di polizia, spedizionieri, investigatori, sindaco, governatore, un database di materiali pericolosi dell'EPA, amministratore di sistema e così via. Dobbiamo poi consolidare questa lista in un piccolo numero di attori, che sono diversi dal punto di vista dell'utilizzo del sistema. Ad esempio, un vigile del fuoco e un agente di polizia possono condividere la stessa interfaccia con il sistema, in quanto entrambi sono coinvolti in un unico incidente sul campo. Un dispatcher, invece, gestisce più incidenti simultanei e richiede l'accesso a una quantità maggiore di informazioni. È probabile che il sindaco e il governatore non interagiscano direttamente con il sistema, ma si avvalgano invece dei servizi di un operatore addestrato.

Una volta identificati gli attori, il passo successivo nell'attività di sollecitazione dei requisiti è quello di determinare le funzionalità che saranno accessibili a ciascun attore. Queste informazioni possono essere estratte utilizzando scenari e formalizzate utilizzando casi d'uso.

4.4.2 Identificare gli scenari

Uno scenario è "una descrizione narrativa di ciò che le persone fanno e sperimentano mentre cercano di utilizzare sistemi e applicazioni informatiche" [Carroll, 1995]. Uno scenario è una descrizione concreta, focalizzata, informale di una singola caratteristica del sistema dal punto di vista di un singolo attore. Gli scenari non possono (e non sono destinati a) sostituire i casi d'uso, in quanto si concentrano su casi specifici ed eventi concreti (al contrario di descrizioni complete e generali). Tuttavia, gli scenari migliorano l'evocazione dei requisiti fornendo uno strumento comprensibile per gli utenti e i clienti.

La Figura 4-6 è un esempio di scenario per il sistema FRIEND, un sistema informativo per la risposta agli incidenti. In questo scenario, un agente di polizia segnala un incendio e un dispaccio avvia la risposta all'incidente. Si noti che questo scenario è concreto, nel senso che

descrive un singolo

Scenario namewarehouseOnFire	
Casi di attori partecipanti	bob, alice:FieldOfficer john:Dispatcher
Flusso di eventi1.	Bob, guidando lungo la strada principale con la sua auto di pattuglia, nota il fumo che esce da un magazzino. La sua compagna, Alice, attiva la funzione "Segnala emergenza" dal suo portatile FRIEND. 2. Alice inserisce l'indirizzo dell'edificio, una breve descrizione della sua posizione (ad esempio, angolo nord-ovest) e un livello di emergenza. Oltre ad un'unità antincendio, richiede diverse unità paramediche sul posto, dato che l'area sembra essere relativamente trafficata. Conferma il suo intervento e attende un riscontro. 3. John, il dispacciatore, viene avvisato dell'emergenza con un bip della sua postazione di lavoro. Egli esamina le informazioni presentate da Alice e prende atto del rapporto. Assegna un'unità antincendio e due unità paramediche al luogo dell'incidente e invia ad Alice il loro orario di arrivo previsto (ETA). 4. Alice riceve il riconoscimento e l'ETA.

Figura 4-6 scenario warehouseOnFire per il caso d'uso ReportEmergency.

esempio. Non cerca di descrivere tutte le possibili situazioni in cui viene segnalato un incidente di incendio. In particolare, gli scenari non possono contenere descrizioni di decisioni. Per descrivere l'esito di una decisione, sarebbero necessari due scenari, uno per il percorso "vero" e uno per il percorso "falso".

Gli scenari possono avere molti usi diversi durante l'evocazione dei requisiti e durante altre attività del ciclo di vita. Di seguito è riportato un numero selezionato di tipi di scenari presi da [Carroll, 1995]:

- **Gli scenari As-is** descrivono una situazione attuale. Durante la reingegnerizzazione, ad esempio, il sistema attuale viene compreso osservando gli utenti e descrivendo le loro azioni come scenari. Questi scenari possono poi essere validati per la correttezza e la precisione con gli utenti.
- **Scenari visionari** descrivono un sistema futuro. Gli scenari visionari sono utilizzati sia come punto nello spazio di modellazione da parte degli sviluppatori mentre perfezionano le loro idee sul sistema futuro, sia come mezzo di comunicazione per sollecitare le esigenze degli utenti. Gli scenari visionari possono essere visti come un prototipo economico.
- **Gli scenari di valutazione** descrivono i compiti dell'utente rispetto ai quali il sistema deve essere valutato. Lo sviluppo collaborativo di scenari di valutazione da parte di utenti e sviluppatori migliora anche la definizione delle funzionalità testate da questi scenari.
- Gli **scenari di formazione** sono tutorial utilizzati per introdurre nuovi utenti al sistema. Si tratta di istruzioni passo dopo passo progettate per tenere a mano l'utente

attraverso compiti comuni.

Nell'elicitazione dei requisiti, gli sviluppatori e gli utenti scrivono e perfezionano una serie di scenari per ottenere una comprensione condivisa di ciò che il sistema dovrebbe essere. Inizialmente, ogni scenario può essere di alto livello e incompleto, come lo scenario warehouseOnFire. Le seguenti domande possono essere utilizzate per identificare gli scenari.

Domande per l'identificazione degli scenari

- Quali sono i compiti che l'attore vuole che il sistema svolga?
- A quali informazioni accede l'attore? Chi crea questi dati? Possono essere modificati o rimossi? Da chi?
- Di quali cambiamenti esterni ha bisogno l'attore per informare il sistema? Con quale frequenza? Quando?
- Di quali eventi ha bisogno il sistema per informare l'attore? Con quale latenza?

Gli sviluppatori utilizzano i documenti esistenti sul dominio applicativo per rispondere a queste domande. Questi documenti includono i manuali d'uso dei sistemi precedenti, i manuali delle procedure, gli standard aziendali, le note utente e i fogli di calcolo, i colloqui con gli utenti e i clienti. Gli sviluppatori dovrebbero sempre scrivere scenari utilizzando i termini del dominio applicativo, in contrapposizione ai propri termini. Man mano che gli sviluppatori acquisiscono una maggiore comprensione del dominio applicativo e delle possibilità della tecnologia disponibile, affinano in modo iterativo e incrementale gli scenari per includere una quantità sempre maggiore di dettagli. Il disegno di modelli di interfaccia utente spesso aiuta a trovare omissioni nelle specifiche e a costruire un quadro più concreto del sistema.

Nell'esempio di FRIEND, identifichiamo quattro scenari che abbracciano il tipo di attività che il sistema dovrebbe supportare:

- warehouseOnFire (Figura 4-6): Un incendio viene rilevato in un magazzino; due agenti sul campo arrivano sul posto e richiedono risorse.
- parafangoBender: Un incidente d'auto senza vittime si verifica in autostrada. Gli agenti di polizia documentano l'incidente e gestiscono il traffico mentre i veicoli danneggiati vengono trainati via.
- catInatree: Un gatto è bloccato su un albero. Un camion dei pompieri viene chiamato per recuperare il gatto. Poiché l'incidente è a bassa priorità, l'autopompa impiega tempo ad arrivare sulla scena. Nel frattempo, l'impaziente proprietario del gatto si arrampica sull'albero, cade e si rompe una gamba, richiedendo l'invio di un'ambulanza.
- Terremoto: Un terremoto senza precedenti danneggia gravemente edifici e strade, provocando molteplici incidenti e innescando l'attivazione di un piano operativo di emergenza in tutto lo Stato. Il governatore viene informato. I danni alle strade ostacolano la risposta agli incidenti.

L'enfasi per gli sviluppatori durante l'identificazione dell'attore e l'identificazione dello scenario è la comprensione del dominio applicativo. Ciò si traduce in una comprensione condivisa della portata del sistema e dei processi di lavoro degli utenti da supportare. Una volta che gli sviluppatori hanno identificato e descritto gli attori e gli scenari, formalizzano gli

scenari in casi d'uso.

4.4.3 Identificare i casi d'uso

Uno **scenario** è un'istanza di un **caso d'uso**, cioè un caso d'uso specifica tutti i possibili scenari per una data funzionalità. Un caso d'uso è avviato da un attore. Dopo la sua iniziazione, un caso d'uso può interagire anche con altri attori. Un caso d'uso rappresenta un flusso completo di eventi attraverso il sistema nel senso che descrive una serie di interazioni correlate che risultano dalla sua iniziazione.

La Figura 4-7 illustra il caso d'uso ReportEmergency di cui lo scenario warehouseOnFire (vedi Figura 4-6) è un'istanza. L'attore FieldOfficer avvia questo caso d'uso attivando la funzione "Report Emergency" di FRIEND. Il caso d'uso si completa quando l'attore FieldOfficer riceve il riconoscimento che è stato creato un incidente. I passi nel flusso degli eventi sono dentellati per indicare chi avvia il passo. I passi 1 e 3 sono iniziati dall'attore, mentre i passi

Utilizzare il nome del caso ReportEmergency	
Attori partecipanti	Iniziato da FieldOfficer Comunica con l'operatore
Flusso di eventi1. Il FieldOfficer attiva la funzione "Segnala emergenza" del suo terminale.	
	2 FRIEND risponde presentando un modulo al FieldOfficer.
	3 Il FieldOfficer compila il modulo selezionando il livello di emergenza, il tipo, la posizione e una breve descrizione della situazione. Il FieldOfficer descrive anche le possibili risposte alla situazione di emergenza. Una volta completato il modulo, il FieldOfficer invia il modulo.
	4 FRIEND riceve il modulo e lo notifica al Dispatcher.
	5 Il Dispatcher esamina le informazioni inviate e crea un Incidente nel database invocando il caso d'uso di OpenIncident. Il Dispatcher seleziona una risposta e riconosce il rapporto.
	6 FRIEND visualizza il riconoscimento e la risposta selezionata al FieldOfficer.
Condizione di ingresso- Il FieldOfficer è loggato in FRIEND.	
Condizioni di uscita - Il FieldOfficer ha ricevuto un riconoscimento e la risposta selezionata dal Dispatcher, OPPURE • Il FieldOfficer ha ricevuto una spiegazione che indica il motivo per cui la transazione non ha potuto essere elaborata.	
Requisiti di qualità	• Il rapporto del FieldOfficer viene riconosciuto entro 30 secondi. • La risposta selezionata arriva non più tardi di 30 secondi dopo l'invio da parte del Spedizioniere.

Figura 4-7 Un esempio di caso d'uso, ReportEmergency. Sotto ReportEmergency, la colonna di sinistra indica le azioni dell'attore, mentre la colonna di destra indica le risposte del sistema.

2 e 4 vengono avviati dal sistema. Questo caso d'uso è generale e comprende una serie di scenari. Ad esempio, il caso d'uso ReportEmergency potrebbe essere applicato anche allo scenario fenderBender. I casi d'uso possono essere scritti a vari livelli di dettaglio come nel caso degli scenari.

Generalizzare gli scenari e identificare i casi d'uso di alto livello che il sistema deve supportare permette agli sviluppatori di definire la portata del sistema. Inizialmente, gli sviluppatori danno un nome ai casi d'uso, li allegano agli attori iniziatori e forniscono una descrizione di alto livello del caso d'uso come nella Figura 4-7. Il nome di un caso d'uso dovrebbe essere una frase verbale che denota ciò che l'attore sta cercando di realizzare. La frase verbale "Report Emergency" indica che un attore sta tentando di segnalare un'emergenza al sistema (e quindi all'attore del Dispatcher). Questo caso d'uso non si chiama "Record Emergency" perché il nome dovrebbe riflettere la prospettiva dell'attore, non del sistema. Inoltre, non è chiamato "Tentativo di segnalare un'emergenza" perché il nome dovrebbe riflettere l'obiettivo del caso d'uso, non l'attività effettiva.

L'allegazione di casi d'uso agli attori iniziatori permette agli sviluppatori di chiarire i ruoli dei diversi utenti. Spesso, concentrandosi su chi avvia ogni caso d'uso, gli sviluppatori identificano nuovi attori che sono stati precedentemente trascurati.

La descrizione di un caso d'uso comporta l'indicazione di quattro campi. Descrivere le condizioni di entrata e di uscita di un caso d'uso permette agli sviluppatori di comprendere le condizioni in cui un caso d'uso viene invocato e l'impatto del caso d'uso sullo stato dell'ambiente e del sistema. Esaminando le condizioni di entrata e di uscita dei casi d'uso, gli sviluppatori possono determinare se ci possono essere casi d'uso mancanti. Ad esempio, se un caso d'uso richiede l'attivazione del piano operativo di emergenza in caso di terremoti, la specifica dei requisiti dovrebbe anche fornire un caso d'uso per l'attivazione di tale piano. La descrizione del flusso di eventi di un caso d'uso consente agli sviluppatori e ai clienti di discutere l'interazione tra gli attori e il sistema. Ciò si traduce in molte decisioni sul confine del sistema, cioè sulla decisione di quali azioni sono compiute dall'attore e quali azioni sono compiute dal sistema. Infine, la descrizione dei requisiti di qualità associati a un caso d'uso permette agli sviluppatori di ottenere requisiti non funzionali nel contesto di una specifica funzionalità. In questo libro, ci concentriamo su questi quattro campi per descrivere i casi d'uso in quanto descrivono gli aspetti più essenziali di un caso d'uso. In pratica, si possono aggiungere molti campi aggiuntivi per descrivere un flusso eccezionale di eventi, regole e invarianti che il caso d'uso deve rispettare durante il flusso degli eventi.

La scrittura dei casi d'uso è un mestiere. Un analista impara a scrivere meglio i casi d'uso con esperienza. Di conseguenza, analisti diversi tendono a sviluppare stili diversi, il che può rendere difficile produrre una specifica coerente dei requisiti. Per affrontare la questione di come imparare a scrivere i casi d'uso e come garantire la coerenza tra i casi d'uso di una specifica dei requisiti, gli analisti adottano una guida alla scrittura dei casi d'uso. La Figura 4-8 è una semplice guida alla scrittura adattata da [Cockburn, 2001] che può essere usata per chi scrive casi d'uso per principianti. La Figura 4-9 fornisce un esempio di un caso d'uso povero

che viola la linea guida per la scrittura in diversi modi.

Il caso d'uso ReportEmergency della Figura 4-7 può essere abbastanza illustrativo per descrivere come FRIEND supporta la segnalazione di emergenze e per ottenere un feedback generale da parte dell'utente, ma non fornisce sufficienti dettagli per una specifica dei requisiti. Successivamente, si discute di come i casi d'uso siano raffinati e dettagliati.

Guida alla scrittura di casi d'uso semplice

- I casi d'uso dovrebbero essere denominati con frasi verbo. Il nome del caso d'uso dovrebbe indicare ciò che l'utente sta cercando di realizzare (ad esempio, ReportEmergency, OpenIncident).
- Gli attori devono essere nominati con le frasi dei nomi (ad es. FieldOfficer, Dispatcher, Victim...).
- Il confine del sistema dovrebbe essere chiaro. I passi compiuti dall'attore e i passi compiuti dal sistema devono essere distinti (ad esempio, nella Figura 4-7, le azioni del sistema sono dentellate a destra).
- Utilizzare i passi del caso nel flusso degli eventi devono essere formulati nella voce attiva. Questo rende esplicito chi ha compiuto il passo.
- Il rapporto di causalità tra le fasi successive deve essere chiaro.
- Un caso d'uso deve descrivere una transazione utente completa (ad esempio, il caso d'uso ReportEmergency descrive tutti i passaggi tra l'avvio della segnalazione d'emergenza e la ricezione di un riconoscimento).
- Le eccezioni devono essere descritte separatamente.
- Un caso d'uso non dovrebbe descrivere l'interfaccia utente del sistema. Questo toglie l'attenzione dai passi effettivi compiuti dall'utente e viene meglio affrontato con mock-up visivi (ad esempio, il ReportEmergency si riferisce solo alla funzione "Report Emergency", non al menu, al pulsante, né al comando effettivo che corrisponde a questa funzione).
- Un caso d'uso non dovrebbe superare le due o tre pagine di lunghezza. Altrimenti, l'uso include ed estende le relazioni per decomporlo in casi d'uso più piccoli, come spiegato nella sezione 4.4.5.

Figura 4-8 Esempio di guida alla scrittura del caso d'uso.

Usa case nameAccidentBad name: Cosa sta cercando di fare l'utente?
Attore inizialeIniziato da FieldOfficer
Flusso di eventi1. Il FieldOfficer segnala l'incidente.
2. Viene inviata un'ambulanza. *Causalità: Quale azione ha fatto sì che il FieldOfficer ricevesse un riconoscimento?*
Voce passiva: Chi manda l'ambulanza?
3. Il Dispatcher viene avvisato quando l'ambulanza arriva sul posto. *Transazione incompleta: Cosa fa il FieldOfficer dopo la spedizione dell'ambulanza?*

Figura 4-9 Un esempio di un caso di cattivo utilizzo. Le violazioni della guida alla scrittura sono indicate in *corsivo* nella colonna di destra.

4.4.4 Raffinazione dei casi d'uso

La Figura 4-10 è una versione raffinata del caso d'uso ReportEmergency. È stata estesa per includere dettagli sul tipo di incidenti noti a FRIEND e interazioni dettagliate che indicano come il Dispatcher riconosce il FieldOfficer.

Utilizzare il nome del casoRapportoEmergenza	
Attori partecipanti	Iniziato da FieldOfficer Comunica con l'operatore

Flusso di eventi1. Il FieldOfficer attiva la funzione "Segnala emergenza" del suo terminale.

2 L'AMICIENTE risponde presentando un modulo all'ufficiale. *Il modulo include un menu di tipo di emergenza (emergenza generale, incendio, trasporto) e la posizione, la descrizione dell'incidente, la richiesta di risorse e i campi relativi ai materiali pericolosi.*

3 Il FieldOfficer compila il modulo *specificando minimamente il tipo di emergenza e i campi di descrizione*. Il FieldOfficer può anche descrivere le possibili risposte alla situazione di emergenza *e richiedere risorse specifiche*. Una volta completato il modulo, il FieldOfficer invia il modulo.

4 FRIEND riceve il modulo e lo notifica al Dispatcher *tramite una finestra di dialogo a comparsa.*

5 Il Dispatcher esamina le informazioni inviate e crea un Incidente nel database invocando il caso d'uso di OpenIncident. *Tutte le informazioni contenute nel modulo del FieldOfficer sono automaticamente incluse nell'Incidente. Il Dispatcher seleziona una risposta assegnando risorse all'Incident (con il caso d'uso AllocateResources) e riconosce il rapporto di emergenza inviando un breve messaggio al FieldOfficer.*

6 FRIEND visualizza il riconoscimento e la risposta selezionata al FieldOfficer.

Condizione di ingresso- ...

Figura 4-10 Descrizione raffinata per il caso d'uso ReportEmergency. Aggiunte sottolineate in *corsivo.*

L'uso di scenari e casi d'uso per definire le funzionalità del sistema mira a creare requisiti che siano convalidati dall'utente fin dalle prime fasi di sviluppo. All'inizio della progettazione e dell'implementazione del sistema, il costo della modifica delle specifiche dei requisiti e dell'aggiunta di nuove funzionalità impreviste aumenta. Anche se i requisiti cambiano fino a tardi nello sviluppo, gli sviluppatori e gli utenti dovrebbero sforzarsi di affrontare la maggior parte dei problemi relativi ai requisiti in anticipo. Questo comporta molti cambiamenti e molta validazione durante la selezione dei requisiti. Si noti che molti casi d'uso vengono riscritti più volte, altri sostanzialmente perfezionati e altri ancora completamente

è caduto. Per risparmiare tempo, gran parte del lavoro di esplorazione può essere fatto utilizzando scenari e modelli di interfaccia utente.

Le seguenti euristiche possono essere utilizzate per scrivere scenari e casi d'uso:

Euristica per lo sviluppo di scenari e casi d'uso

- Utilizzare scenari per comunicare con gli utenti e per convalidare le funzionalità.
- In primo luogo, perfezionare un singolo scenario per comprendere le ipotesi dell'utente sul sistema. L'utente può avere familiarità con sistemi simili, nel qual caso l'adozione di specifiche convenzioni di interfaccia utente renderebbe il sistema più utilizzabile.
- Successivamente, definire molti scenari non molto dettagliati per definire la portata del sistema. Convalidare con l'utente.
- Utilizzare i mock-up solo come supporto visivo; la progettazione dell'interfaccia utente dovrebbe avvenire come compito separato dopo che la funzionalità è sufficientemente stabile.
- Presentare all'utente alternative multiple e molto diverse (invece di estrarre un'unica alternativa dall'utente). La valutazione di alternative diverse amplia l'orizzonte dell'utente. La generazione di alternative diverse costringe gli sviluppatori a "pensare fuori dagli schemi".
- Dettagliare un'ampia fetta verticale quando la portata del sistema e le preferenze dell'utente sono ben comprese. Convalidare con l'utente.

Il fulcro di questa attività è la completezza e la correttezza. Gli sviluppatori identificano le funzionalità non coperte dagli scenari e le documentano perfezionando i casi d'uso o scrivendo nuovi casi d'uso. Gli sviluppatori descrivono casi che si verificano raramente e la gestione delle eccezioni così come sono visti dagli attori. Mentre l'identificazione iniziale dei casi d'uso e degli attori si è concentrata sulla definizione dei confini del sistema, il perfezionamento dei casi d'uso fornisce sempre più dettagli sulle caratteristiche fornite dal sistema e sui vincoli ad essi associati. In particolare, i seguenti aspetti dei casi d'uso, inizialmente ignorati, sono dettagliati durante il perfezionamento:

- Gli elementi che vengono manipolati dal sistema sono dettagliati. Nella Figura 4-10, abbiamo aggiunto dettagli sugli attributi del modulo di segnalazione di emergenza e sui tipi di incidenti.
- Viene specificata la sequenza a basso livello delle interazioni tra l'attore e il sistema. Nella Figura 4-10, abbiamo aggiunto informazioni su come il `Dispatcher` genera un riconoscimento selezionando le risorse.
- I diritti di accesso (che gli attori possono invocare in quali casi d'uso) sono specificati.
- Le eccezioni mancanti sono identificate e la loro gestione è specificata.
- Le funzionalità comuni tra i casi d'uso sono state eliminate.

Nella sezione successiva, descriviamo come riorganizzare gli attori e utilizzare i casi con le relazioni, che affronta gli ultimi tre punti di cui sopra.

4.4.5 Identificare le relazioni tra gli attori e i casi d'uso

Anche i sistemi di medie dimensioni hanno molti casi d'uso. Le relazioni tra gli attori e i casi d'uso consentono agli sviluppatori e agli utenti di ridurre la complessità del modello e di aumentarne la comprensibilità. Utilizziamo le relazioni di comunicazione tra gli attori e i casi d'uso per descrivere il sistema in livelli di funzionalità. Utilizziamo relazioni estese per separare i flussi di eventi eccezionali e comuni. Utilizziamo le relazioni per ridurre la ridondanza tra i casi d'uso.

Relazioni di comunicazione tra gli attori e casi d'uso

Le relazioni di comunicazione tra gli attori e i casi d'uso rappresentano il flusso di informazioni durante il caso d'uso. L'attore che avvia il caso d'uso deve essere distinto dagli altri attori con cui il caso d'uso comunica. Specificando quale attore può invocare uno specifico use case, specifichiamo implicitamente anche quali attori non possono invocare l'use case. Allo stesso modo, specificando quali attori comunicano con uno specifico use case, specifichiamo quali attori possono accedere a specifiche informazioni e quali no. Così, documentando le relazioni di iniziazione e di comunicazione tra gli attori e i casi d'uso, specifichiamo il controllo dell'accesso al sistema a un livello grossolano.

Le relazioni tra gli attori e i casi d'uso sono identificate quando vengono identificati i casi d'uso. La Figura 4-11 illustra un esempio di relazioni di comunicazione nel caso del sistema FRIEND. Lo stereotipo "inizia" denota l'avvio del caso d'uso da parte di un attore, e lo stereotipo "partecipa" denota che un attore (che non ha avviato il caso d'uso) comunica con il caso d'uso.

Estendere le relazioni tra i casi d'uso

Un caso d'uso estende un altro caso d'uso se il caso d'uso esteso può includere il comportamento dell'estensione in determinate condizioni. Nell'esempio FRIEND, si supponga che la connessione tra la stazione FieldOfficer e la stazione Dispatcher sia interrotta mentre l'opzione

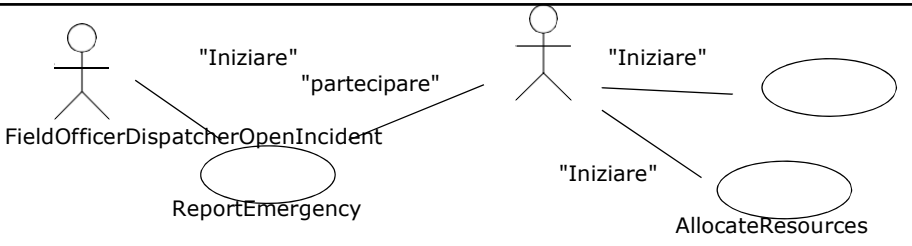


Figura 4-11 Esempio di relazioni di comunicazione tra attori e casi d'uso in FRIEND (diagramma dei casi d'uso UML). Il FieldOfficer avvia il caso d'uso ReportEmergency e il Dispatcher avvia i casi d'uso OpenIncident e AllocateResources. I FieldOfficer non possono aprire direttamente un incidente o allocare risorse da soli.

FieldOfficer sta compilando il modulo (ad esempio, l'auto del FieldOfficer entra in una galleria). La stazione del FieldOfficer deve notificare al FieldOfficer che il suo modulo non è stato consegnato e quali misure deve prendere. Il caso d'uso ConnectionDown è modellato come un'estensione di ReportEmergency (vedi Figura 4-12). Le condizioni in cui il caso d'uso di ConnectionDown viene avviato sono descritte in ConnectionDown in contrapposizione a ReportEmergency. La separazione dei flussi di eventi eccezionali e opzionali dal caso d'uso di base presenta due vantaggi. In primo luogo, il caso d'uso di base diventa più breve e più facile da comprendere. In secondo luogo, il caso comune si distingue dal caso eccezionale, che consente agli sviluppatori di trattare ogni tipo di funzionalità in modo diverso (ad esempio, ottimizzare il caso comune per il tempo di risposta, ottimizzare il caso eccezionale per la robustezza). Sia il caso d'uso esteso che le estensioni sono casi d'uso completi. Ciascuno di essi deve avere condizioni di ingresso e di fine ed essere comprensibile per l'utente come un insieme indipendente.

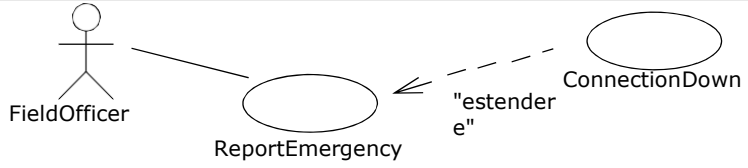


Figura 4-12 Esempio di utilizzo della relazione di estensione (diagramma di caso d'uso UML). ConnectionDown estende il caso d'uso ReportEmergency. Il caso d'uso ReportEmergency diventa più breve e si concentra esclusivamente sulla segnalazione di emergenza.

Includere le relazioni tra i casi d'uso

Le ridondanze tra i casi d'uso possono essere calcolate utilizzando anche le relazioni. Si supponga, ad esempio, che un Dispacciatore debba consultare la mappa della città quando si apre un incidente (ad esempio, per valutare quali aree sono a rischio durante un incendio) e quando si assegnano risorse (ad esempio, per trovare quali risorse sono più vicine all'incidente). In questo caso, il caso d'uso della ViewMap descrive il flusso di eventi richiesto quando si visualizza la mappa della città ed è utilizzato sia dal caso d'uso dell'OpenIncident che da quello dell'AllocateResources (Figura 4-13).

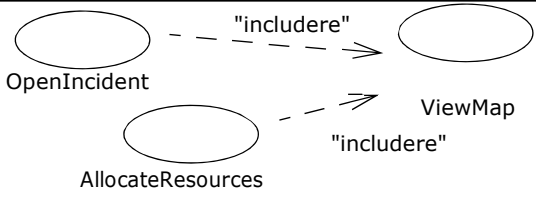


Figura 4-13 Esempio di includere le relazioni tra i casi d'uso. ViewMap descrive il flusso di eventi per la visualizzazione di una mappa di una città (ad esempio, scorrimento, zoom, interrogazione per nome della via) ed è usato sia da casi d'uso OpenIncident che da casi d'uso AllocateResources.

L'eliminazione dei comportamenti condivisi dai casi d'uso ha molti vantaggi, tra cui descrizioni più brevi e meno ridondanze. Il comportamento dovrebbe essere considerato in un caso d'uso separato *solo* se è condiviso tra due o più casi d'uso. L'eccessiva frammentazione delle specifiche dei requisiti in un gran numero di casi d'uso rende le specifiche confuse per gli utenti e i clienti.

Estendere contro includere le relazioni

Includere ed estendere sono costrutti simili, e all'inizio potrebbe non essere chiaro allo sviluppatore quando usare ciascuno di essi [Jacobson et al., 1992]. La principale distinzione tra questi costrutti è la direzione della relazione. Per includere le relazioni, l'evento che fa scattare il caso d'uso target (cioè, incluso) è descritto nel flusso di eventi del caso d'uso sorgente. Per le relazioni di estensione, l'evento che fa scattare il caso d'uso della sorgente (cioè l'estensione) è descritto nel caso d'uso della sorgente come precondizione. In altre parole, per le relazioni di inclusione, ogni caso d'uso incluso deve specificare dove il caso d'uso incluso deve essere invocato. Per le relazioni di estensione, solo il caso d'uso di estensione specifica quali casi d'uso sono estesi. Quindi, un comportamento che è fortemente legato a un evento e che si verifica solo in un numero relativamente limitato di casi d'uso dovrebbe essere rappresentato con una relazione inclusa. Questi tipi di comportamento di solito includono funzioni di sistema comuni che possono essere utilizzate in diversi luoghi (ad esempio, la visualizzazione di una mappa, la specificazione di un nome di file, la selezione di un elemento). Al contrario, un comportamento che può accadere in qualsiasi momento o il cui verificarsi può essere più facilmente specificato come condizione di ingresso dovrebbe essere rappresentato con una relazione estesa. Questi tipi di comportamento comprendono situazioni eccezionali (ad esempio, invocare la guida in linea, annullare una transazione, gestire un guasto della rete).

La Figura 4-14 mostra l'esempio di ConnectionDown descritto con una relazione di inclusione (colonna di sinistra) e con una relazione di estensione (colonna di destra). Nella colonna di sinistra è necessario inserire il testo in due punti del flusso di eventi in cui si può invocare il caso d'uso ConnectionDown. Inoltre, se vengono descritte ulteriori situazioni eccezionali (ad esempio, una funzione di aiuto sulla stazione FieldOfficer), il caso d'uso ReportEmergency dovrà essere modificato e diventerà ingombrante di condizioni. Nella colonna di destra, dobbiamo descrivere solo le condizioni in cui viene invocato il caso d'uso eccezionale, che può includere un gran numero di casi d'uso (ad esempio, "qualsiasi caso d'uso in cui si perde la connessione tra il FieldOfficer e il Dispatcher"). Inoltre, possono essere aggiunte ulteriori situazioni eccezionali senza modificare il caso d'uso di base (ad esempio, ReportEmergency). La possibilità di estendere il sistema senza modificare le parti esistenti è fondamentale, in quanto ci permette di garantire che il comportamento originale venga lasciato inalterato. La distinzione tra includere ed estendere è un problema di documentazione: l'utilizzo del tipo di relazione corretta riduce le dipendenze tra i casi d'uso, riduce la ridondanza e abbassa la probabilità di introdurre errori quando i requisiti cambiano. Tuttavia, l'impatto sulle altre attività di sviluppo è minimo.

In sintesi, le seguenti euristiche possono essere utilizzate per la selezione di un rapporto di estensione o di inclusione.

Euristica per estendere e includere le relazioni

- Utilizzare relazioni estese per comportamenti eccezionali, facoltativi o raramente ricorrenti. Un esempio di comportamento raramente ricorrente è la rottura di una risorsa (ad esempio, un camion dei pompieri). Un esempio di comportamento opzionale è la notifica di risorse vicine che rispondono ad un incidente non correlato.
- L'uso include relazioni per comportamenti condivisi in due o più casi d'uso.
- Tuttavia, utilizzare la discrezione quando si applicano le due euristiche di cui sopra e non sovrastrutturare il modello di caso d'uso. Alcuni casi d'uso più lunghi (ad es. due pagine) sono più facili da capire e da rivedere rispetto a molti casi brevi (ad es. dieci righe).

In tutti i casi, lo scopo dell'aggiunta di includere ed estendere le relazioni è quello di ridurre o eliminare le ridondanze dal modello dei casi d'uso, eliminando così potenziali incongruenze.

4.4.6 Identificazione degli oggetti di analisi iniziale

Uno dei primi ostacoli che gli sviluppatori e gli utenti incontrano quando iniziano a collaborare tra loro è la diversa terminologia. Anche se alla fine gli sviluppatori imparano la terminologia degli utenti, è probabile che questo problema si ripresenti quando si aggiungono nuovi sviluppatori al progetto. Le incomprensioni derivano dall'uso degli stessi termini in contesti diversi e con significati diversi.

Per stabilire una terminologia chiara, gli sviluppatori identificano gli **oggetti partecipanti** per ogni caso d'uso. Gli sviluppatori dovrebbero identificarli, nominarli e descriverli senza ambiguità e raccoglierli in un glossario. [3] La costruzione di questo glossario costituisce il primo passo verso l'analisi, che noi
discutere nel prossimo capitolo.

Il glossario è incluso nella specifica dei requisiti e, successivamente, nei manuali d'uso. Gli sviluppatori tengono aggiornato il glossario in base all'evoluzione delle specifiche dei requisiti. I vantaggi del glossario sono molteplici: i nuovi sviluppatori sono esposti ad un insieme coerente di definizioni, un singolo termine è usato per ogni concetto (invece di un termine per sviluppatori e un termine per utenti), e ogni termine ha un preciso e chiaro significato ufficiale.

L'identificazione degli oggetti partecipanti risulta nel modello oggetto di analisi iniziale. L'identificazione degli oggetti partecipanti durante la selezione dei requisiti costituisce solo un primo passo verso il modello di oggetti di analisi completo. Il modello di analisi completo non viene solitamente utilizzato come mezzo di comunicazione tra utenti e sviluppatori, poiché gli utenti non hanno spesso familiarità con i concetti orientati agli oggetti. Tuttavia, la descrizione degli oggetti (cioè le definizioni dei termini nel glossario) e i loro attributi sono visibili agli utenti e rivisti. Descriviamo in dettaglio l'ulteriore perfezionamento del modello di analisi nel capitolo 5, *Analisi*.

3. Il glossario è anche chiamato "dizionario dei dati" [Rumbaugh et al., 1991].

RelazioneEmergenza (includere la relazione)	ReportEmergenza (estendere il rapporto)
1. ... 2. ... 3. Il FieldOfficer compila il modulo per selezionando il livello di emergenza, il tipo, la posizione, e una breve descrizione della situazione. Il sito FieldOfficer descrive anche possibili risposte alla situazione di emergenza. Una volta che il il modulo viene compilato, il FieldOfficer invia la forma, a quel punto, il Dispacciatore è notificato. *Se il collegamento con il Dispatcher è rotto, viene utilizzato il caso d'uso ConnectionDown.* 4. Se il collegamento è ancora attivo, il Dispatcher esamina le informazioni presentate e crea un Incidente nel database invocando l'opzione Caso d'uso OpenIncident. Il dispacciatore seleziona una risposta e riconosce il rapporto di emergenza. *Se il collegamento è interrotto, il Si utilizza il caso d'uso ConnectionDown.* 5. ...	1. ... 2. ... 3. Il FieldOfficer compila il modulo per selezionando il livello di emergenza, il tipo, la posizione, e una breve descrizione della situazione. Il sito FieldOfficer descrive anche possibili risposte alla situazione di emergenza. Una volta che il il modulo viene compilato, il FieldOfficer invia la forma, a quel punto, il Dispacciatore è notificato. 4. 4. Il Dispacciatore esamina il informazioni e crea un Incidente nel invocando l'uso di OpenIncident caso. Il Dispatcher seleziona una risposta e prende atto del rapporto d'emergenza. 5. ...
ConnectionDown (include la relazione)	ConnectionDown (prolungare il rapporto)
	Il caso d'uso ConnectionDown estende qualsiasi caso d'uso in cui la comunicazione tra il FieldOfficer e il Dispatcher può andare persa.
1 Il FieldOfficer e il Dispatcher vengono informati che il collegamento è interrotto. Essi vengono informati dei possibili motivi per cui un tale evento si verificherebbe (ad esempio, "La stazione del FieldOfficer è in una galleria?"). 2 La situazione viene registrata dal sistema e recuperata quando la connessione viene ristabilita. 3 Il FieldOfficer e il Dispatcher entrano in contatto attraverso altri mezzi e il Dispatcher avvia il ReportEmergency dalla stazione del Dispatcher.	1 Il FieldOfficer e il Dispatcher vengono informati che il collegamento è interrotto. Essi vengono informati dei possibili motivi per cui un tale evento si verificherebbe (ad esempio, "La stazione del FieldOfficer è in una galleria?"). 2 La situazione viene registrata dal sistema e recuperata quando la connessione viene ristabilita. 3 Il FieldOfficer e il Dispatcher entrano in contatto attraverso altri mezzi e il Dispatcher avvia il ReportEmergency dalla stazione del Dispatcher.

Figura 4-14 Aggiunta della condizione eccezionale di ConnectionDown a ReportEmergency. Una relazione estesa viene utilizzata per il flusso di eventi eccezionali e facoltativi perché fornisce una descrizione più modulare.

Nella letteratura sono state proposte molte euristiche per l'identificazione degli oggetti. Eccone alcune selezionate:

Euristica per l'identificazione degli oggetti di analisi iniziale

- Termini che gli sviluppatori o gli utenti devono chiarire per comprendere il caso d'uso
- Sostantivi ricorrenti nei casi d'uso (ad esempio, Incidente)
- Entità del mondo reale che il sistema deve monitorare (ad esempio, FieldOfficer, Risorse)
- Processi del mondo reale che il sistema deve tenere traccia (ad es. EmergencyOperationsPlan)
- Casi d'uso (ad esempio, ReportEmergency)
- Fonti di dati o lavelli (ad es. stampante)
- Artefatti con cui l'utente interagisce (ad esempio, Stazione)
- Utilizzare *sempre i* termini del dominio dell'applicazione.

Durante la selezione dei requisiti, per ogni caso d'uso vengono generati oggetti partecipanti. Se due casi d'uso si riferiscono allo stesso concetto, l'oggetto corrispondente dovrebbe essere lo stesso. Se due oggetti condividono lo stesso nome e non corrispondono allo stesso concetto, uno o entrambi i concetti vengono rinominati per riconoscere ed enfatizzare la loro differenza. Questo consolidamento elimina ogni ambiguità nella terminologia utilizzata. Per esempio, la Tabella 4-2 illustra gli oggetti partecipanti iniziali che abbiamo identificato per il caso d'uso ReportEmergency.

Tabella 4-2 Oggetti **partecipanti** per il caso d'uso ReportEmergency.

Ufficiale **di Polizia**	che gestisce gli Incidenti. Un Dispatcher apre, documenta e chiude gli incidenti in risposta ai rapporti di emergenza e ad altre comunicazioni con gli agenti sul campo. I Dispatcher sono identificati da numeri di badge.
EmergencyReport	Rapporto iniziale su un incidente da un FieldOfficer a un Dispatcher. Un rapporto EmergencyReport solitamente fa scattare la creazione di un Incidente da parte del Dispatcher. Un EmergencyReport è composto da un livello di emergenza, un tipo (incendio, incidente stradale, altro), una località e una descrizione.
FieldOfficer	Police o ufficiale dei vigili del fuoco in servizio. Un FieldOfficer può essere assegnato ad un solo incidente alla volta. I FieldOfficer sono identificati da numeri di distintivo.
IncidenteSituazione	che richiede l'attenzione di un agente sul campo. Un Incidente può essere segnalato nel sistema da un FieldOfficer o da chiunque altro esterno al sistema. Un Incidente è composto da una descrizione, una risposta, uno stato (aperto, chiuso, documentato), una località e un numero di FieldOfficer.

Una volta identificati e consolidati gli oggetti partecipanti, gli sviluppatori possono utilizzarli come lista di controllo per assicurarsi che il set di casi d'uso identificati sia completo.

Euristica per il controllo incrociato dei casi d'uso e degli oggetti partecipanti

- Quali casi d'uso creano questo oggetto (cioè, durante quali casi d'uso sono i valori degli attributi dell'oggetto inseriti nel sistema)?
- Quali attori possono accedere a queste informazioni?
- Quali casi d'uso modificano e distruggono questo oggetto (cioè, quali casi d'uso modificano o rimuovono queste informazioni dal sistema)?
- Quale attore può avviare questi casi d'uso?
- Questo oggetto è necessario (cioè, c'è almeno un caso d'uso che dipende da queste informazioni?)

4.4.7 Identificazione dei requisiti non funzionali

I requisiti non funzionali descrivono aspetti del sistema che non sono direttamente correlati al suo comportamento funzionale. I requisiti non funzionali si estendono su una serie di questioni, dall'aspetto dell'interfaccia utente ai requisiti di tempo di risposta ai problemi di sicurezza. I requisiti non funzionali sono definiti contemporaneamente ai requisiti funzionali perché hanno lo stesso impatto sullo sviluppo e sul costo del sistema.

Per esempio, si consideri un display a mosaico che un controllore del traffico aereo utilizza per tracciare gli aerei. Un sistema di visualizzazione a mosaico compila i dati da una serie di radar e database (da cui il termine "mosaico") in un display di sintesi che indica tutti gli aerei in una certa area, compresa la loro identificazione, velocità e altitudine. Il numero di aerei che un tale sistema può visualizzare limita le prestazioni del controllore del traffico aereo e il costo del sistema. Se il sistema può gestire solo pochi aerei contemporaneamente, il sistema non può essere utilizzato in aeroporti affollati. D'altra parte, un sistema in grado di gestire un gran numero di aerei è più costoso e più complesso da costruire e da testare.

I requisiti non funzionali possono avere un impatto inaspettato sul lavoro dell'utente. Per ottenere con precisione tutti i requisiti essenziali non funzionali, sia il cliente che lo sviluppatore devono collaborare in modo da identificare (minimamente) quali attributi del sistema che sono difficili da realizzare sono critici per il lavoro dell'utente. Nell'esempio del display a mosaico di cui sopra, il numero di aerei che un singolo display a mosaico deve essere in grado di gestire ha implicazioni sulla dimensione delle icone utilizzate per la visualizzazione degli aerei, sulle caratteristiche per l'identificazione degli aerei e le loro proprietà, sulla frequenza di aggiornamento dei dati, e così via.

L'insieme di requisiti non funzionali che ne risulta include tipicamente requisiti contrastanti. Per esempio, i requisiti non funzionali del SatWatch (Figura 4-3) richiedono un meccanismo accurato, in modo che l'ora non debba mai essere azzerata, ed un basso costo unitario, in modo che sia accettabile per l'utente sostituire l'orologio con uno nuovo quando si rompe. Questi due requisiti non funzionali sono in conflitto, poiché il costo unitario dell'orologio aumenta con la sua precisione. Per affrontare tali conflitti, il cliente e lo sviluppatore danno priorità ai requisiti non funzionali, in modo che possano essere affrontati in modo coerente durante la realizzazione del sistema.

Tabella 4-3Domande **esemplificative** per la determinazione di requisiti non funzionali.

Domande Category	esemplificative
Usabilità - Qual è il livello di competenza dell'utente?	• Quali standard di interfaccia utente sono familiari all'utente? • Quale documentazione deve essere fornita all'utente?
Affidabilità *(compresa la robustezza, la sicurezza e la protezione)*	• Quanto deve essere affidabile, disponibile e robusto il sistema? • Il riavvio del sistema è accettabile in caso di guasto? • Quanti dati può perdere il sistema? • Come deve gestire il sistema le eccezioni? • Ci sono requisiti di sicurezza del sistema? • Ci sono requisiti di sicurezza del sistema?
Prestazioni - Quanto deve essere reattivo il sistema?	• Le attività degli utenti sono critiche dal punto di vista del tempo? • Quanti utenti contemporanei dovrebbe supportare? • Quanto è grande un tipico archivio dati per sistemi comparabili? • Qual è la latenza peggiore che è accettabile per gli utenti?
Sostenibilità *(compresa la manutenibilità e la portabilità)*	• Quali sono le estensioni previste per il sistema? • Chi si occupa della manutenzione del sistema? • Ci sono piani per il porting del sistema in diversi ambienti software o hardware?
Implementazione- Ci sono dei vincoli sulla piattaforma hardware?	• I vincoli sono imposti dal team di manutenzione? • I vincoli sono imposti dal team di prova?
Interfaccia- Il sistema deve interagire con i sistemi esistenti?	• Come vengono esportati/importati i dati nel sistema? • Quali standard in uso da parte del cliente dovrebbero essere supportati dal sistema?
Funzionamento- Chi gestisce il sistema in funzione?	
Imballaggio- Chi installa il sistema?	• Quanti impianti sono previsti? • Ci sono vincoli di tempo per l'installazione?
Legale - Come deve essere concesso in licenza il sistema?	• I problemi di responsabilità sono associati a guasti del sistema? • Le royalty o i costi di licenza sono dovuti all'utilizzo di specifici algoritmi o componenti?

Ci sono purtroppo pochi metodi sistematici per ottenere requisiti non funzionali. In pratica, gli analisti usano una tassonomia dei requisiti non funzionali (ad esempio, lo schema FURPS+ descritto in precedenza) per generare check list di domande per aiutare il cliente e gli sviluppatori a concentrarsi sugli aspetti non funzionali del sistema. Poiché gli attori del sistema sono già stati identificati a questo punto, questa check list può essere organizzata per ruolo e distribuita agli utenti rappresentativi. Il vantaggio di tali check list è che possono essere riutilizzate e ampliate per ogni nuovo sistema in un determinato dominio applicativo, riducendo così il numero di omissioni. Si noti che tali check list possono anche comportare l'ottenimento di ulteriori requisiti funzionali. Ad esempio, quando si pongono domande sul funzionamento del sistema, il cliente e gli sviluppatori possono scoprire una serie di casi d'uso legati all'amministrazione del sistema. La tabella 4-3 illustra le domande di esempio per ciascuna delle categorie FURPS+.

Una volta che il cliente e gli sviluppatori identificano un insieme di requisiti non funzionali, possono organizzarli in grafici di affinamento e di dipendenza per identificare ulteriori requisiti non funzionali e identificare i conflitti. Per ulteriore materiale su questo argomento, si rimanda il lettore alla letteratura specializzata (ad esempio, [Chung et al., 1999]).

4.5 Gestione dei requisiti Elicitazione

Nella sezione precedente abbiamo descritto gli aspetti tecnici della modellazione di un sistema in termini di casi d'uso. La modellazione dei casi d'uso da sola, tuttavia, non costituisce di per sé un'elicitazione dei requisiti. Anche dopo essere diventati esperti nella modellazione dei casi d'uso, gli sviluppatori devono comunque sollecitare i requisiti dagli utenti e trovare un accordo con il cliente. In questa sezione descriviamo i metodi per ottenere informazioni dagli utenti e negoziare un accordo con il cliente. In particolare, descriviamo:

- Negoziazione delle specifiche con i clienti: Progettazione dell'applicazione congiunta (Sezione 4.5.1)
- Mantenere la tracciabilità (Sezione 4.5.2)
- Elicitazione dei requisiti di documentazione (Sezione 4.5.3).

4.5.1 Negoziare le specifiche con i clienti: Progettazione di applicazioni congiunte

Il **Joint Application Design (JAD)** è un metodo di requisiti sviluppato presso IBM alla fine degli anni '70. La sua efficacia è data dal fatto che il lavoro di selezione dei requisiti viene svolto in un'unica sessione di workshop a cui partecipano tutte le parti interessate. Utenti, clienti, sviluppatori e un leader di sessione addestrato siedono insieme in una stanza per presentare i loro punti di vista, ascoltare altri punti di vista, negoziare e giungere a una soluzione reciprocamente accettabile. Il risultato del workshop, il documento finale del JAD, è un documento completo sulle specifiche dei requisiti che include le definizioni degli elementi dei dati, i flussi di lavoro e le schermate di interfaccia. Poiché il documento finale è sviluppato congiuntamente dalle parti interessate (cioè i partecipanti che non solo hanno interesse al successo del progetto, ma possono anche prendere decisioni sostanziali), il documento JAD finale rappresenta un accordo tra utenti, clienti e sviluppatori, e quindi riduce al minimo i cambiamenti dei requisiti in seguito nel processo di sviluppo. JAD è composto da cinque

attività (Figura 4-15):

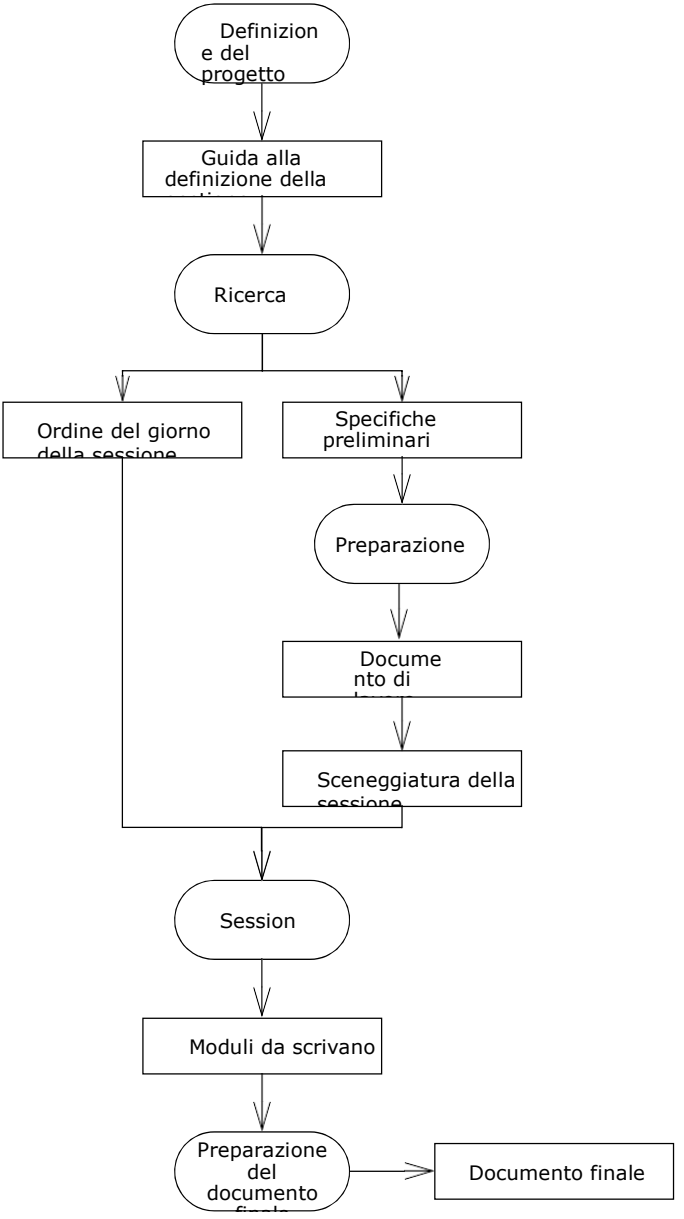


Figura 4-15 Attività di JAD (diagramma di attività UML). Il cuore di JAD è l'attività di Sessione durante la quale tutti gli stakeholder progettano e concordano una specifica dei requisiti. Le attività che precedono la Sessione massimizzano la sua efficienza. La produzione del documento finale cattura le decisioni prese durante la Sessione.

1. *Definizione del progetto*. Durante questa attività, il facilitatore JAD intervista il project manager e il cliente per determinare gli obiettivi e la portata del progetto. I risultati delle interviste sono raccolti nella *Guida alla definizione del management*.
2. *Ricerca*. Durante questa attività, il facilitatore JAD intervista gli utenti presenti e futuri, raccoglie informazioni sul dominio applicativo e descrive una prima serie di casi d'uso di alto livello. Il JAD facilitator inizia anche un elenco di problemi da affrontare durante la sessione. I risultati di questa attività sono un'*Agenda della sessione* e una *Specifica preliminare che* elenca il flusso di lavoro e le informazioni sul sistema.
3. *Preparazione*. Durante questa attività, il facilitatore JAD prepara la sessione. Il facilitatore JAD crea un *documento di lavoro*, che è la prima bozza del documento finale, un'agenda per la sessione ed eventuali diapositive o lavagne a fogli mobili che rappresentano le informazioni raccolte durante l'attività di ricerca. Il facilitatore JAD seleziona anche un team composto dal cliente, dal project manager, dagli utenti selezionati e dagli sviluppatori. Tutti gli stakeholder sono rappresentati e i partecipanti sono in grado di prendere decisioni vincolanti.
4. *Sessione*. Durante questa attività, il facilitatore JAD guida il team nella creazione delle specifiche dei requisiti. Una sessione JAD dura dai 3 ai 5 giorni. Il team definisce e concorda gli scenari, i casi d'uso e i modelli di interfaccia utente. Tutte le decisioni sono documentate da uno scriba.
5. *Documento finale*. Il facilitatore JAD prepara il *Documento finale*, rivedendo il documento di lavoro per includere tutte le decisioni prese durante la sessione. Il Documento finale rappresenta una specifica completa del sistema concordato durante la sessione. Il Documento finale viene distribuito ai partecipanti alla sessione per la revisione. I partecipanti partecipano quindi a una riunione di 1 o 2 ore per discutere le revisioni e finalizzare il documento.

JAD è stato utilizzato da IBM e da altre aziende. JAD sfrutta le dinamiche di gruppo per migliorare la comunicazione tra i partecipanti e per accelerare il consenso. Al termine di una sessione JAD, gli sviluppatori sono più consapevoli delle esigenze degli utenti e gli utenti sono più consapevoli dei compromessi di sviluppo. Ulteriori vantaggi derivano da una riduzione delle attività di riprogettazione a valle. A causa della sua dipendenza dalle dinamiche sociali, il successo di una sessione JAD dipende spesso dalle qualifiche del facilitatore JAD come facilitatore di incontri. Per una panoramica dettagliata del JAD, si rimanda al lettore [Wood & Silver, 1989].

4.5.2 Mantenere la tracciabilità

La tracciabilità è la capacità di seguire la vita di un'esigenza. Ciò include il tracciamento da dove provengono i requisiti (ad esempio, chi li ha generati, a quale necessità del cliente si rivolge) a quali aspetti del sistema e del progetto interessato (ad esempio, quali componenti realizzano il requisito, quale test case verifica la sua realizzazione). La tracciabilità consente agli sviluppatori di mostrare che il sistema è completo, ai tester di dimostrare che il sistema è

conforme ai suoi requisiti, ai progettisti di registrare la logica alla base del sistema e ai manutentori di valutare l'impatto del cambiamento.

Consideriamo il sistema SatWatch che abbiamo introdotto all'inizio del capitolo. Attualmente, la specifica richiede un display a due righe che includa l'ora e la data. Dopo che il cliente decide che la dimensione delle cifre è troppo piccola per una comoda lettura, gli sviluppatori cambiano il requisito di visualizzazione in un display a una sola riga combinato con un pulsante per passare da ora e data. La tracciabilità ci permetterebbe di rispondere alle seguenti domande:

- Chi ha originato il requisito di visualizzazione a due righe?
- Qualche vincolo implicito ha imposto questo requisito?
- Quali componenti devono essere modificati a causa del pulsante e del display aggiuntivi?
- Quali casi di test devono essere modificati?

L'approccio più semplice per mantenere la tracciabilità è quello di utilizzare riferimenti incrociati tra documenti, modelli e artefatti del codice. Ogni singolo elemento (ad esempio, requisito, componente, classe, operazione, test case) è identificato da un numero unico. Le dipendenze sono poi documentate manualmente come un riferimento incrociato testuale contenente il numero dell'elemento sorgente e il numero dell'elemento di destinazione. Il supporto dello strumento può essere semplice come un foglio di calcolo o uno strumento di elaborazione testi. Questo approccio è costoso in termini di tempo e di personale ed è soggetto ad errori. Tuttavia, per i piccoli progetti, gli sviluppatori possono osservare i benefici in anticipo.

Per progetti su larga scala, strumenti di database specializzati consentono l'automazione parziale dell'acquisizione, la modifica e il collegamento delle dipendenze di tracciabilità ad un livello più dettagliato (ad esempio, DOORS [Telelogic] o RequisitePro [Rational]). Tali strumenti riducono il costo del mantenimento della tracciabilità, ma richiedono il buy-in e la formazione della maggior parte degli stakeholder e impongono restrizioni ad altri strumenti nel processo di sviluppo.

4.5.3 Requisiti di documentazione Elicitazione

I risultati dell'elaborazione dei requisiti e delle attività di analisi sono documentati nel **documento di analisi dei requisiti (RAD)**. Questo documento descrive completamente il sistema in termini di requisiti funzionali e non funzionali. Il pubblico del RAD include il cliente, gli utenti, la gestione del progetto, gli analisti di sistema (cioè gli sviluppatori che partecipano ai requisiti) e i progettisti di sistema (cioè gli sviluppatori che partecipano alla progettazione del sistema). La prima parte del documento, che include i casi d'uso e i requisiti non funzionali, è scritta durante la selezione dei requisiti. La formalizzazione delle specifiche in termini di modelli di oggetti è scritta durante l'analisi. La Figura 4-16 è un modello di esempio per un RAD usato in questo libro.

La prima sezione del RAD è un'*introduzione*. Il suo scopo è quello di fornire una breve panoramica della funzione del sistema e delle ragioni del suo sviluppo, della sua portata e dei riferimenti al contesto di sviluppo (ad esempio, riferimento alla dichiarazione del problema scritta dal cliente, riferimenti ai sistemi esistenti, studi di fattibilità). L'introduzione comprende anche gli obiettivi e i criteri di successo del progetto.

La seconda sezione, *Sistema attuale*, descrive lo stato attuale della situazione. Se il nuovo

sistema sostituirà un sistema esistente, questa sezione descrive le funzionalità e i problemi

Documento di analisi dei requisiti

1. Introduzione
   - 1.1 Scopo del sistema
   - 1.2 Portata del sistema
   - 1.3 Obiettivi e criteri di successo del progetto
   - 1.4 Definizioni, acronimi e abbreviazioni
   - 1.5 Riferimenti
   - 1.6 Panoramica
2. Sistema attuale
3. Sistema proposto
   - 3.1 Panoramica
   - 3.2 Requisiti funzionali
   - 3.3 Requisiti non funzionali
     - 3.3.1 Usabilità
     - 3.3.2 Affidabilità
     - 3.3.3 Prestazioni
     - 3.3.4 Sostenibilità
     - 3.3.5 Implementazione
     - 3.3.6 Interfaccia
     - 3.3.7 Imballaggio
     - 3.3.8 Legale
   - 3.4 Modelli di sistema
     - 3.4.1 Scenari
     - 3.4.2 Modello di valigetta
     - *3.4.3 Modello dell'oggetto*
     - *3.4.4 Modello dinamico*
     - 3.4.5 Interfaccia utente - Percorsi di navigazione e screen mock-up
4. Glossario

Figura 4-16 Schema del documento di analisi dei requisiti (RAD). Le sezioni in *corsivo* sono completate durante l'analisi (vedi capitolo successivo).

del sistema attuale. Per il resto, questa sezione descrive come vengono svolti ora i compiti supportati dal nuovo sistema. Ad esempio, nel caso di `SatWatch`, l'utente attualmente resetta l'orologio ogni volta che viaggia attraverso un fuso orario. A causa della natura manuale di questa operazione, l'utente imposta occasionalmente l'ora sbagliata e occasionalmente trascura di effettuare il reset. Al contrario, il `SatWatch` assicura continuamente un'ora precisa nel corso della sua vita. Nel caso di `FRIEND`, il sistema attuale è basato sulla carta: gli spedizionieri tengono traccia delle assegnazioni delle risorse compilando dei moduli. La comunicazione tra i dispatcher e gli agenti sul campo avviene via radio. Il sistema attuale richiede un elevato costo di documentazione e di gestione che `FRIEND` mira a ridurre.

La terza sezione, *Sistema proposto,* documenta l'elicitazione dei requisiti e il modello di analisi del nuovo sistema. È divisa in quattro sottosezioni:

- *Panoramica* presenta una panoramica funzionale del sistema.

- *I requisiti funzionali* descrivono l'alto livello di funzionalità del sistema.
- *I requisiti non funzionali* descrivono i requisiti a livello utente che non sono direttamente correlati alla funzionalità. Ciò include l'usabilità, l'affidabilità, le prestazioni, il supporto, l'implementazione, l'interfaccia, il funzionamento, l'imballaggio e i requisiti legali.
- *I modelli di sistema* descrivono gli scenari, i casi d'uso, il modello a oggetti e i modelli dinamici del sistema. Questa sezione contiene le specifiche funzionali complete, compresi i modelli che illustrano l'interfaccia utente del sistema e i percorsi di navigazione che rappresentano la sequenza delle schermate. Le sottosezioni *Modello a oggetti* e *Modello dinamico* sono scritte durante l'attività di Analisi, descritta nel prossimo capitolo.

Il RAD dovrebbe essere scritto dopo che il modello del caso d'uso è stabile, cioè quando il numero di modifiche ai requisiti è minimo. I requisiti, tuttavia, vengono aggiornati durante tutto il processo di sviluppo quando vengono scoperti problemi di specifica o quando viene cambiato il campo di applicazione del sistema. Il RAD, una volta pubblicato, viene baselineato e messo sotto
gestione della configurazione. [4] La sezione della cronologia delle revisioni del RAD fornirà una cronologia di
I cambiamenti includono l'autore responsabile di ogni cambiamento, la data del cambiamento e una breve descrizione del cambiamento.

4.6 Caso di studio ARENA

In questa sezione applichiamo al sistema ARENA i concetti e i metodi descritti in questo capitolo. Iniziamo con la dichiarazione iniziale del problema fornita dal cliente, e sviluppiamo un modello di caso d'uso e un modello di oggetto di analisi iniziale. Nelle sezioni precedenti abbiamo selezionato degli esempi per il loro valore illustrativo. In questa sezione ci concentriamo su un esempio realistico, descriviamo gli artefatti man mano che vengono creati e perfezionati. Questo ci permette di discutere di compromessi più realistici e decisioni di progettazione e di concentrarci su dettagli operativi che in genere non sono visibili negli esempi illustrativi. In questa discussione, "ARENA" denota il sistema in generale, mentre "arena" denota una specifica istanziazione del sistema.

4.6.1 Dichiarazione iniziale del problema

Dopo un primo incontro con il cliente, viene scritta la dichiarazione del problema (Figura 4-17).

Si noti che questo breve testo descrive il problema e i requisiti ad alto livello. Questa non è tipicamente la fase in cui ci impegniamo a rispettare un budget o una data di consegna. Per prima cosa, iniziamo a sviluppare il modello dei casi d'uso identificando attori e scenari.

4. Una **baseline** è una versione di un prodotto di lavoro che è stata rivista e formalmente approvata. La

gestione della configurazione è il processo di tracciamento e approvazione delle modifiche alla baseline. Parliamo della gestione della configurazione nel Capitolo 13, *Gestione della configurazione*.

Dichiarazione sul problema ARENA

1. Problema

La popolarità di Internet e del World Wide Web ha permesso la creazione di una varietà di comunità virtuali, gruppi di persone che condividono interessi comuni, ma che non si sono mai incontrate di persona. Tali comunità virtuali possono essere di breve durata (ad esempio, un gruppo di persone che si incontrano in una chat room o che partecipano a un torneo) o di lunga durata (ad esempio, gli iscritti a una mailing list). Possono includere un piccolo gruppo di persone o molte migliaia.

Molti giochi per computer multi-player ora includono il supporto per le comunità virtuali che sono giocatori del gioco dato. I giocatori possono ricevere notizie sugli aggiornamenti del gioco, nuove mappe e personaggi; possono annunciare e organizzare le partite, confrontare i punteggi e scambiare consigli. La società di giochi sfrutta questa infrastruttura per generare entrate o per pubblicizzare i propri prodotti.

Attualmente, tuttavia, ogni società di giochi sviluppa un tale supporto comunitario in ogni singolo gioco. Ogni azienda utilizza un'infrastruttura diversa, concetti diversi e fornisce diversi livelli di supporto. Questa ridondanza e incoerenza si traduce in molti svantaggi, tra cui una curva di apprendimento per i giocatori quando entrano a far parte di ogni nuova comunità, per le società di gioco che devono sviluppare il supporto da zero e per gli inserzionisti che devono contattare ogni singola comunità separatamente. Inoltre, questa soluzione non offre molte opportunità di fertilizzazione incrociata tra le diverse comunità.

2. Obiettivi

Gli obiettivi del progetto ARENA sono i seguenti:

- fornire un'infrastruttura per la gestione di un'arena, compresa la registrazione di nuove partite e giocatori, l'organizzazione di tornei e il controllo dei punteggi dei giocatori.
- fornire un quadro di riferimento per i proprietari del campionato per personalizzare il numero e la sequenza delle partite e l'accumulo dei punti di valutazione degli esperti.
- fornire un quadro di riferimento per gli sviluppatori di giochi per lo sviluppo di nuovi giochi o per l'adattamento dei giochi esistenti nel quadro di ARENA.
- fornire un'infrastruttura per gli inserzionisti.

3. Requisiti funzionali

ARENA supporta cinque tipi di utenti:

- L'*operatore* dovrebbe essere in grado di definire nuove partite, definire nuovi stili di torneo (ad esempio, tornei ad eliminazione diretta, campionati, best of series), definire nuove formule di rating per esperti e gestire gli utenti.
- *I proprietari della lega* dovrebbero essere in grado di definire un nuovo campionato, organizzare e annunciare nuovi tornei all'interno di una lega, condurre un torneo e dichiarare un vincitore.
- *I giocatori* dovrebbero essere in grado di iscriversi in un'arena, fare domanda per un campionato, giocare le partite che sono assegnate al giocatore, o abbandonare il torneo.
- *Gli spettatori* dovrebbero essere in grado di monitorare ogni partita in corso e controllare i punteggi e le statistiche delle partite e dei giocatori passati. Gli spettatori non hanno bisogno di registrarsi in un'arena.
- L'*inserzionista* dovrebbe essere in grado di caricare nuovi annunci, selezionare uno schema pubblicitario (ad esempio, sponsor del torneo, sponsor della lega), controllare il saldo dovuto e cancellare gli annunci.

Figura 4-17 Dichiarazione iniziale del problema ARENA.

4. Requisiti non funzionali

- *Basso costo di gestione*. L'operatore deve essere in grado di installare e amministrare un'arena senza l'acquisto di componenti software aggiuntivi e senza l'aiuto di un amministratore di sistema a tempo pieno.
- *Estensibilità*. L'operatore deve essere in grado di aggiungere nuovi giochi, nuovi stili di torneo e nuove formule di valutazione per esperti. Tali aggiunte possono richiedere lo spegnimento temporaneo del sistema e l'aggiunta di nuovi moduli (ad es. classi Java). Tuttavia, non dovrebbe essere richiesta alcuna modifica del sistema esistente.
- *Scalabilità*. Il sistema deve supportare il calcio d'inizio di molti tornei paralleli (ad esempio, 10), ognuno dei quali coinvolge fino a 64 giocatori e diverse centinaia di spettatori simultanei.
- *Rete a bassa larghezza di banda*. I giocatori dovrebbero essere in grado di giocare partite tramite un modem analogico a 56K o più velocemente.

5. Ambiente di destinazione

- Tutti gli utenti dovrebbero essere in grado di accedere a qualsiasi arena con un browser web che supporti i cookie, Javascript e applet Java. Le funzioni di amministrazione (ad esempio, l'aggiunta di nuovi giochi, stili di torneo e utenti) utilizzate dall'operatore non dovrebbero essere disponibili sul web.
- ARENA dovrebbe funzionare su qualsiasi sistema operativo Unix (ad esempio MacOS X, Linux, Solaris).

Figura 4-17 *Continua*.

4.6.2 Identificare gli attori e gli scenari

Identifichiamo cinque attori, uno per ogni tipo di utente nella dichiarazione del problema (Operatore, Titolare della Lega, Giocatore, Spettatore e Inserzionista). Poiché la funzionalità principale del sistema è quella di organizzare e giocare i tornei, sviluppiamo prima di tutto uno scenario di esempio, organizziamo il TicTacToeTournament (Figura 4-18) per suscitare ed esplorare questa funzionalità in modo più dettagliato. Concentrandoci prima su una stretta fetta verticale del sistema, comprendiamo meglio le aspettative del cliente sul sistema, compresi i confini del sistema e i tipi di interazione tra l'utente e il sistema. Utilizzando lo scenario organizeTicTacToeTournament della Figura 4-18, produciamo una serie di domande per il cliente raffigurato in (Figura 4-19). Sulla base delle risposte del cliente, perfezioniamo lo scenario di conseguenza.

Si noti che quando si pongono domande a un cliente, il nostro obiettivo primario è quello di comprendere le esigenze del cliente e il dominio applicativo. Una volta compreso il dominio e prodotta una prima versione delle specifiche dei requisiti, possiamo iniziare a negoziare le caratteristiche e i costi con il cliente e dare priorità ai requisiti. Tuttavia, l'intreccio tra l'elicitazione e la negoziazione troppo presto è di solito controproducente.

Dopo aver affinato il primo scenario al punto che entrambi concordiamo con il cliente sul confine del sistema (per quello scenario), ci concentriamo sulla portata complessiva del sistema. Questo viene fatto identificando una serie di scenari più brevi per ogni attore. Inizialmente, questi scenari non sono dettagliati, ma coprono invece un'ampia gamma di funzionalità (Figura 4-20).

Quando incontriamo disaccordi o ambiguità, dettagliamo ulteriormente gli scenari

specifici. In questo esempio, gli scenari definisconoKnockOutStyle e installTicTacToeGame sarebbero affinati ad un livello di dettaglio paragonabile a quello dell'organizeTicTacToeTournament (Figura 4-18).

Scenario nome	*organizeTicTacToeTournament*
Casi di attori partecipanti	alice:Operatore, joe:LeagueOwner, bill:Spettatore, mary:Player
Flusso di eventi	1. Joe, un amico di Alice, è un appassionato di Tic Tac Toe e si offre volontario per organizzare un torneo. 2. Alice registra Joe nell'arena come proprietario di una lega. 3. Joe definisce prima di tutto un campionato per principianti Tic Tac Toe, in cui possono essere ammessi tutti i giocatori. Questo campionato, dedicato alle partite di Tic Tac Toe, prevede che i tornei disputati in questo campionato seguano lo stile del torneo ad eliminazione diretta e la formula "Winner Takes All". 4. Joe organizza il primo torneo del campionato per 16 giocatori a partire dal giorno successivo. 5. Joe annuncia il torneo in una varietà di forum sul web e invia la posta agli altri membri della comunità Tic Tac Toe. 6. Bill e Mary ricevono la notifica via e-mail. 7. Mary è interessata a giocare il torneo e si registra. Altri 19 si candidano. 8. Joe programma 16 giocatori per il torneo e respinge i 4 che si sono applicati per ultimi. 9. I 16 giocatori, tra cui Mary, ricevono un gettone elettronico per l'iscrizione al torneo e l'ora della loro prima partita. 10. Gli altri iscritti alla mailing list Tic Tac Toe, tra cui Bill, ricevono un secondo avviso sul Torneo, con il nome dei giocatori e il calendario delle partite. 11. Al calcio d'inizio del torneo, i giocatori hanno un tempo limitato per partecipare alla partita. Se un giocatore non si presenta, perde la partita. 12. Mary gioca la sua prima partita e vince. Avanza nel torneo ed è prevista per la prossima partita contro un altro vincitore del primo turno. 13. Dopo aver visitato la home page del Torneo Tic Tac Toe, Bill si accorge della vittoria di Mary e decide di assistere alla sua prossima partita. Seleziona la partita e vede la sequenza delle mosse di ogni giocatore nel momento in cui si verificano. Vede anche un banner pubblicitario in fondo al suo browser, che pubblicizza altri tornei e prodotti Tic Tac Toe. 14. Il torneo continua fino all'ultima partita, a quel punto il vincitore del torneo viene dichiarato e al suo record di campionato vengono accreditati tutti i punti associati al torneo. 15. Inoltre, il vincitore del torneo accumula punti per esperti. 16. Joe può scegliere di programmare più tornei nel campionato, nel qual caso i giocatori conosciuti vengono informati della data e viene data priorità ai nuovi giocatori.

Figura 4-18 scenario organizeTicTacToeTournamentale per ARENA.

Scenari tipici, una volta raffinati, si estendono su diverse pagine di testo. Cominciamo anche a mantenere un glossario dei termini importanti, per garantire la coerenza delle specifiche e per assicurare l'utilizzo dei termini del cliente. Ci rendiamo subito conto che i termini Partita, Gioco, Torneo e Lega rappresentano concetti di dominio applicativo che devono essere definiti con precisione, in quanto questi termini potrebbero avere un'interpretazione diversa in altri contesti di gioco. Per realizzare questo, manteniamo un glossario di lavoro e rivediamo le nostre definizioni man mano che il nostro lavoro esplorativo procede (Tabella 4-4).

Fasi 2, 7: Diversi attori si registrano nel sistema. Nel primo caso, l'amministratore registra Joe come proprietario di una lega; nel secondo caso, un giocatore si registra nel sistema.

- La registrazione degli utenti dovrebbe seguire lo stesso paradigma. Chi fornisce le informazioni di registrazione e come vengono esaminate, convalidate e accettate le informazioni?
- *Cliente: Due processi sono confusi nelle fasi 2 e 7, il processo di registrazione, durante il quale i nuovi utenti (ad esempio, un giocatore o il proprietario di una lega) stabiliscono la loro identità, e il processo di richiesta, durante il quale i giocatori indicano di voler partecipare ad un torneo specifico. Durante il processo di registrazione, l'utente fornisce informazioni su se stesso (nome, nickname, e-mail) e sui suoi interessi (tipi di giochi e tornei di cui vuole essere informato). Le informazioni vengono convalidate dall'operatore. Durante il processo di iscrizione, i giocatori indicano a quale torneo vogliono partecipare. Questo viene utilizzato dal proprietario della lega durante la programmazione delle partite.*
- Poiché le informazioni sui giocatori sono già state convalidate dall'operatore, la programmazione delle partite deve essere completamente automatizzata?
- *Cliente: Sì, certo.*

Fase 5: Joe invia la posta ai membri della comunità Tic Tac Toe:

- ARENA offre agli utenti la possibilità di iscriversi a mailing list individuali?
- *Cliente: Sì. Dovrebbero esserci mailing list per annunciare nuove partite, nuovi campionati, nuovi tornei, ecc.*
- ARENA memorizza un profilo utente (ad es. gioco guardato, giochi giocati, interessi specificati da un sondaggio tra gli utenti) a scopo pubblicitario?
- *Cliente: Sì, ma gli utenti dovrebbero comunque essere in grado di registrarsi senza completare un sondaggio, se lo desiderano. Dovrebbero essere incoraggiati a partecipare al sondaggio, ma questo non dovrebbe impedirgli di farlo. Saranno comunque esposti alla pubblicità.*
- Il profilo deve essere utilizzato per iscriversi automaticamente alle mailing list?
- *Cliente: No, pensiamo che gli utenti della nostra comunità preferirebbero avere il controllo completo sulle loro iscrizioni alla mailing list. Immaginare le iscrizioni non darebbe loro l'impressione di avere il controllo.*

Fase 13: Bill sfoglia le statistiche delle partite e decide di vedere la prossima partita in tempo reale.

- Come vengono identificati i giocatori agli spettatori? Per nome reale, per e-mail, per nickname?
- *Cliente: Questo dovrebbe essere lasciato all'utente durante la registrazione.*
- Uno spettatore può riprodurre le vecchie partite?
- *Cliente: I giochi dovrebbero essere in grado di fornire questa capacità, ma alcuni giochi (ad esempio, giochi in tempo reale, giochi d'azione in 3D) potrebbero scegliere di non farlo a causa dei limiti delle risorse.*
- ARENA dovrebbe supportare i giochi in tempo reale?
- *Cliente: Si', questi rappresentano la quota maggiore del nostro mercato. In generale, ARENA dovrebbe supportare la più ampia gamma di giochi possibile.*
- ...

Figura 4-19 Domande generate dallo scenario della Figura 4-18. Risposte del cliente sottolineate in *corsivo*. L'intervistatore può porre domande di follow-up, poiché le nuove conoscenze sono state accidentalmente scoperte per caso.

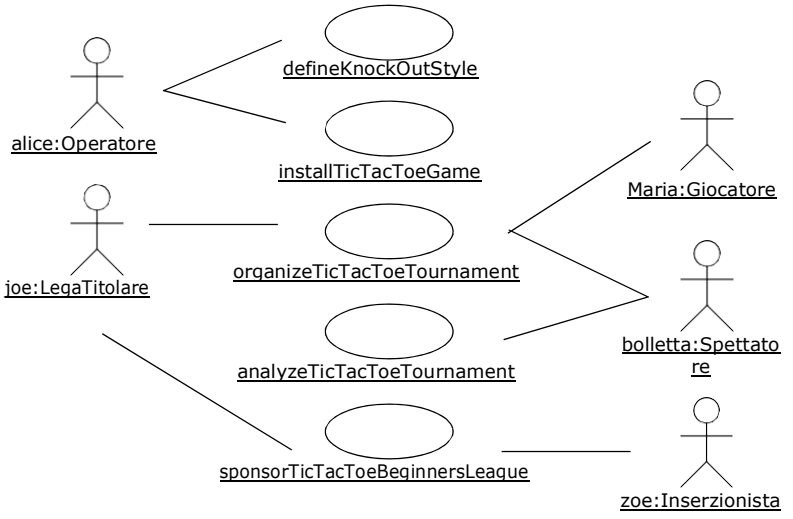


Figura 4-20Scenari di **alto livello** individuati per ARENA. I clienti e gli sviluppatori descrivono inizialmente brevemente gli scenari. Li perfezionano ulteriormente per chiarire ambiguità o per scoprire disaccordi.

Tabella 4-4Glossario di **lavoro** per ARENA. Tenere traccia dei termini importanti e delle loro definizioni garantisce la coerenza delle specifiche e assicura che gli sviluppatori utilizzino il linguaggio del cliente.

GameA	Game è una competizione tra più Giocatori che si svolge secondo un insieme di regole. In ARENA, il termine Gioco si riferisce ad un software che fa rispettare le regole, traccia i progressi di ogni Giocatore e decide il vincitore. Per esempio, il tic tac toe e gli scacchi sono giochi.
MatchA	Match è una gara tra due o più giocatori che seguono le regole di un gioco. Il risultato di un Match può essere un singolo vincitore e un insieme di perdenti o un pareggio (in cui non ci sono vincitori o perdenti). Alcuni Giochi possono non consentire il pareggio.
Torne	**o** Un Torneo è una serie di incontri tra un gruppo di giocatori. I tornei si concludono con un solo vincitore. Il modo in cui i Giocatori accumulano punti e gli Incontri sono programmati è dettato dalla Lega in cui il Torneo è organizzato.
La **LeagueA**	League rappresenta una comunità per la gestione dei tornei. Una League è associata ad un gioco e ad uno stile di torneo specifico. I giocatori iscritti alla Lega accumulano punti secondo la valutazione degli esperti definita nella Lega. Per esempio, una lega scacchistica per principianti ha una formula ExpertRating diversa da quella di una lega per esperti.
TorneoStile	Il TorneoStile definisce il numero di Partite e la loro sequenza per un determinato set di Giocatori. Ad esempio, i giocatori affrontano tutti gli altri giocatori del torneo esattamente una volta in un Round Robin TournamentStyle.

Una volta concordato con il cliente un ambito generale del sistema, formalizziamo le conoscenze acquisite finora sotto forma di casi d'uso di alto livello.

4.6.3 Identificare i casi d'uso

La generalizzazione degli scenari in casi d'uso permette agli sviluppatori di fare un passo indietro rispetto alle situazioni concrete e di considerare il caso generale. Gli sviluppatori possono quindi consolidare le funzionalità correlate in casi d'uso singoli e suddividere le funzionalità non correlate in più casi d'uso.

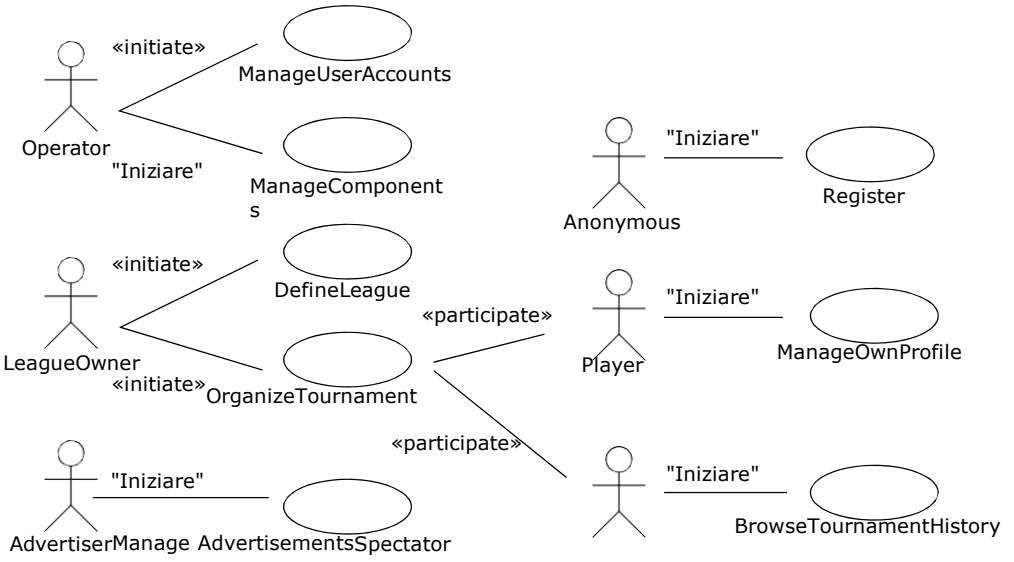
Quando esaminiamo da vicino lo scenario organizeTicTacToeTournament, ci rendiamo conto che esso copre un'ampia gamma di funzionalità avviate da molti attori. Prevediamo che generalizzando questo scenario si otterrebbe un caso d'uso di diverse decine di pagine, e cerchiamo di suddividerlo in casi d'uso autonomi e indipendenti avviati da singoli attori. Per prima cosa decidiamo di dividere le funzionalità relative agli account utente in due casi d'uso, ManageUserAccounts, iniziati dall'Operatore, e Register, iniziati da potenziali giocatori e proprietari della lega (Figura 4-21). Identifichiamo un nuovo attore, Anonymous, che rappresenta questi potenziali utenti che non hanno ancora un account. Allo stesso modo, abbiamo diviso la funzionalità con la navigazione delle partite passate e con la gestione dei profili utente in casi d'uso separati (BrowseTournamentHistory e ManageOwnProfile, iniziati rispettivamente dallo Spectator e dal Player). Infine, per abbreviare ulteriormente il caso d'uso OrganizzazioneTorneo, abbiamo suddiviso la funzionalità per la creazione di nuovi campionati nel caso d'uso DefineLeague, in quanto un LeagueOwner può creare molti tornei nell'ambito di un singolo campionato. Al contrario, prevediamo che l'installazione di nuove partite e nuovi stili richieda passi simili da parte dell'Operatore. Quindi, consolidiamo tutte le funzionalità relative all'installazione di nuovi componenti nel caso d'uso ManageComponents iniziato dall'Operatore.

Catturiamo queste decisioni disegnando un diagramma generale dei casi d'uso e descrivendo brevemente ogni caso d'uso (Figura 4-21). Si noti che un diagramma dei casi d'uso da solo non descrive molte funzionalità. Al contrario, è un indice delle molte descrizioni prodotte durante questa fase. Successivamente, descriviamo i campi di ogni caso d'uso di alto livello, compresi gli attori partecipanti, condizioni di entrata e di uscita, e un flusso di eventi . La Figura 4-22 mostra alto livello OrganizzazioneCaso d'uso del torneo.

Si noti che tutte le fasi di questo flusso di eventi descrivono le azioni degli attori. I casi d'uso di alto livello si concentrano principalmente sul compito svolto dall'attore. L'interazione dettagliata con il sistema, e le decisioni sui confini del sistema, sono inizialmente rimandate alla fase di affinamento. Questo ci permette di descrivere prima il dominio applicativo con i casi d'uso, catturando, in particolare, il modo in cui i diversi attori collaborano per raggiungere i loro obiettivi.

In Figure 4-22, we describe the sequence of actions that are performed by four actors to organize a tournament: the LeagueOwner, who facilitates the complete activity, the Advertiser, to resolve exclusive sponsorship issues, the potential Players who want to participate, and the Spectators. In the first step, we describe the handling of the sponsorship issue, thus making

clear that any sponsorship issue needs to be resolved before the tournament is advertised and before the players apply for the tournament. Originally, the sponsorship issue was not described clearly in the scenarios of Figure 4-20 (which only described the sponsorships of leagues). After



Register Anonymous users register with an Arena for a Player or a League– Owner account. User accounts are required before applying for a tournament or organizing a league. Spectatorsdo not need accounts.

ManageUserAccounts The Operator accepts registrations from LeagueOwners and for Players, cancels existing accounts, and interacts with users about extending their accounts.

ManageComponents The Operator installs new games and defines new tournament styles (generalizes defineKnockOutStyleand installTicTacToeGame).

DefineLeague The LeagueOwner defines a new league (generalizes the first steps of the scenario organizeTicTacToeTournament).

OrganizeTournament The LeagueOwner creates and announces a new tournament, accepts player applications, schedules matches, and kicks off the tournament. During the tournament, players play matches and spectators follow matches. At the end of the tournament, players are credited with points (generalizes the scenario organizeTicTacToeTournament).

ManageAdvertisements The Advertiser uploads banners and sponsors league or tournaments (generalizes sponsorTicTacToeBeginnersLeague).

ManageOwnProfile The Players manage their subscriptions to mailing lists and answer a marketing survey.

BrowseTournamentHistory Spectators examine tournament statistics and player statistics, and replay matches that have already been concluded (generalizes the scenario analyzeTicTacToeTournament).

Figure 4-21 High-level use cases identified for ARENA.

Use case name OrganizeTournament	
Participating actors	Initiated by LeagueOwner Communicates with Advertiser, Player, and Spectator
Flow of events	1. The LeagueOwner creates a Tournament, solicits sponsorships from Advertisers, and announces the Tournament (include use case AnnounceTournament). 2 The Players apply for the Tournament (include use case ApplyForTournament). 3 The LeagueOwner processes the Player applications and assigns them to matches (include use case ProcessApplications). 4 The LeagueOwner kicks off the Tournament (include use case KickoffTournament). 5 The Players compete in the matches as scheduled and Spectators view the matches (include use case PlayMatch). 6 The LeagueOwner declares the winner and archives the Tournament (include use case ArchiveTournament).
Entry condition	• The LeagueOwner is logged into ARENA.
Exit conditions	• The LeagueOwner archived a new tournament in the ARENA archive and the winner has accumulated new points in the league, OR • The LeagueOwner cancelled the tournament and the players' standing in the league is unchanged.

Figure 4-22 An example of a high-level use case, OrganizeTournament.

discussions with the client, we decided to handle also tournament sponsorship, and to handle it at the beginning of each tournament. On the one hand, this enables new sponsors to be added to the system, and on the other hand, it allows the sponsor, in exchange, to advertise the tournament using his or her own resources. Finally, this enables the system to better select advertisement banners during the application process.

In this high-level use case, we boiled down the essentials of the organize–TicTacToeTournament scenario into six steps and left the details to the detailed use case. By describing each high-level use case in this manner, we capture all relationships among actors that the system must be aware of. This also results in a summary description of the system that is understandable to any newcomer to the project.

Next, we write the detailed use cases to specify the interactions between the actors and the system.

4.6.4 Refining Use Cases and Identifying Relationships

Refining use cases enables developers to define precisely the information exchanged among the actors and between the actors and the system. Refining use cases also enables the discovery of alternative flows of events and exceptions that the system should handle.

To keep the case study manageable, we do not show the complete refinement. We start by identifying one detailed use case for each step of the flow of events in the high-level OrganizeTournament use case. The resulting use case diagram is shown in Figure 4-23. We then focus on the use case, AnnounceTournament: Figure 4-24 contains a description of the flow of events, and Figure 4-25 identifies the exceptions that could occur in AnnounceTournament. The remaining use cases will be developed similarly.

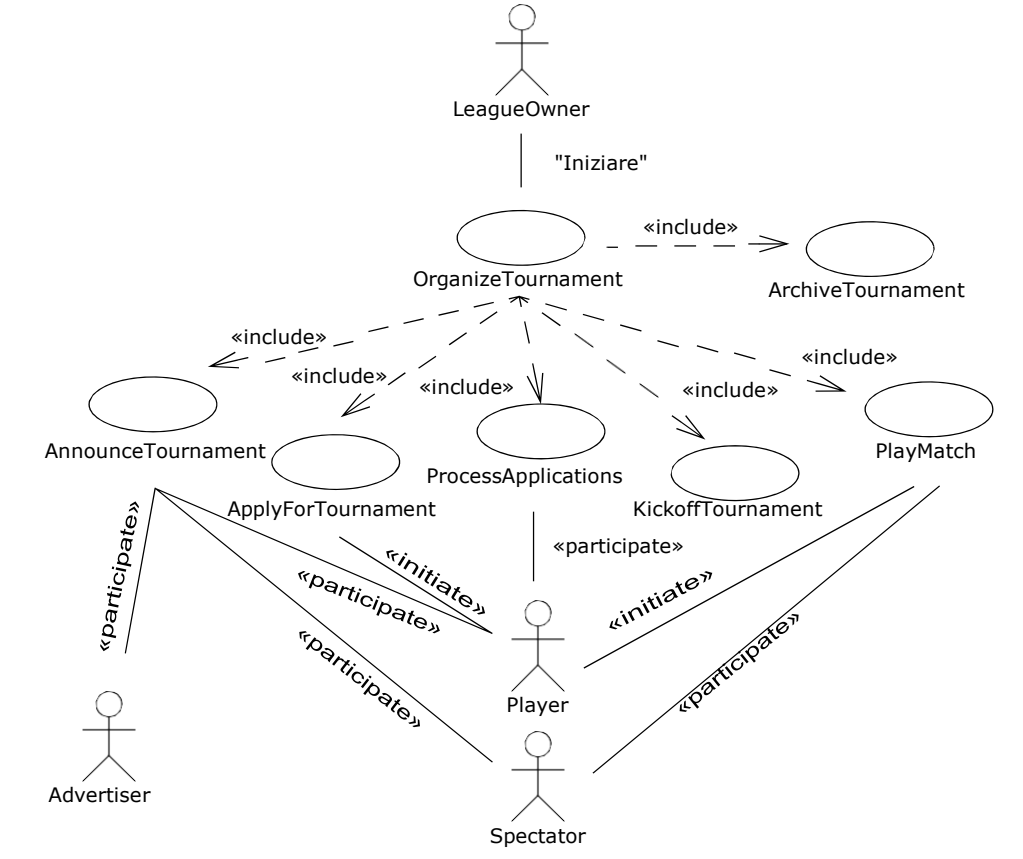


Figure 4-23 Detailed use cases refining the OrganizeTournament high-level use case.

All of the use cases in Figure 4-23 are initiated by the LeagueOwner, except that the ApplyForTournament and PlayMatch are initiated by the Player. The Advertiser participates in AnnounceTournament and the Spectator participates in AnnounceTournament and PlayMatch use cases. The Player participates in all use cases that refine OrganizeTournament. To keep the use case diagram readable, we omitted the «initiate» relationships between the LeagueOwner and the refined use cases. When using a UML modeling tool, we would include those

Name AnnounceTournament	
Participating actors	Initiated by LeagueOwner Communicates with Player, Advertiser, Spectator

Flow of events

1. The LeagueOwner requests the creation of a tournament.
2. The system checks if the LeagueOwner has exceeded the number of tournaments in the league or in the arena. If not, the system presents the LeagueOwner with a form.
3. The LeagueOwner specifies a name, application start and end dates during which Players can apply to the tournament, start and end dates for conducting the tournament, and a maximum number of Players.
4. The system asks the LeagueOwner whether an exclusive sponsorship should be sought and, if yes, presents a list of Advertisers who expressed the desire to be exclusive sponsors.
5. If the LeagueOwner decides to seek an exclusive sponsor, he selects a subset of the names of the proposed sponsors.
6. The system notifies the selected sponsors about the upcoming tournament and the flat fee for exclusive sponsorships.
7. The system communicates their answers to the LeagueOwner.
8. If there are interested sponsors, the LeagueOwner selects one of them.
9. The system records the name of the exclusive sponsor and charges the flat fee for sponsorships to the Advertiser's account. From now on, all advertisement banners associated with the tournament are provided by the exclusive sponsor only.
10. Otherwise, if no sponsors were selected (either because no Advertiser was interested or the LeagueOwner did not select one), the advertisement banners are selected at random and charged to the Advertiser's account on a per unit basis.
11. Once the sponsorship issue is closed, the system prompts the LeagueOwner with a list of groups of Players, Spectators, and Advertisers that could be interested in the new tournament.
12. The LeagueOwner selects which groups to notify.
13. The system creates a home page in the arena for the tournament. This page is used as an entry point to the tournament (e.g., to provide interested Players with a form to apply for the tournament, and to interest Spectators in watching matches).
14. On the application start date, the system notifies each interested user by sending them a link to the main tournament page. The Players can then apply for the tournament with the ApplyForTournament use case until the application end date.

Figure 4-24 An example of a detailed use case, AnnounceTournament.

Entry condition	• The LeagueOwneris logged into ARENA.
Exit conditions	• The sponsorship of the tournament is settled: either a single exclusive Advertiser paid a flat fee or banners are drawn at random from the common advertising pool of the Arena. • Potential Playersreceived a notice concerning the upcoming tournament and can apply for participation. • Potential Spectatorsreceived a notice concerning the upcoming tournament and know when the tournament is about to start. • The tournament home page is available for any to see, hence, other potential Spectatorscan find the tournament home page via web search engines, or by browsing the Arenahome page.
Quality requirements	• Offers to and replies from Advertisersrequire secure authentication, so that Advertiserscan be billed solely on their replies. • Advertisersshould be able to cancel sponsorship agreements within a fixed period, as required by local laws.

Figure 4-24 *Continued.*

relationships as well. We start by writing out the flow of events for the AnnounceTournament use case (Figure 4-24).

The steps in Figure 4-24 describe in detail the information exchanged between the actor and the system. Note, however, that we did not describe any details of the user interface (e.g., forms, buttons, layout of windows or web pages). It is much easier to design a usable user interface later, after we know the intent and responsibility of each actor. Hence, the focus on the refinement phase is to assign (or discover) the detailed intent and responsibilities of each actor.

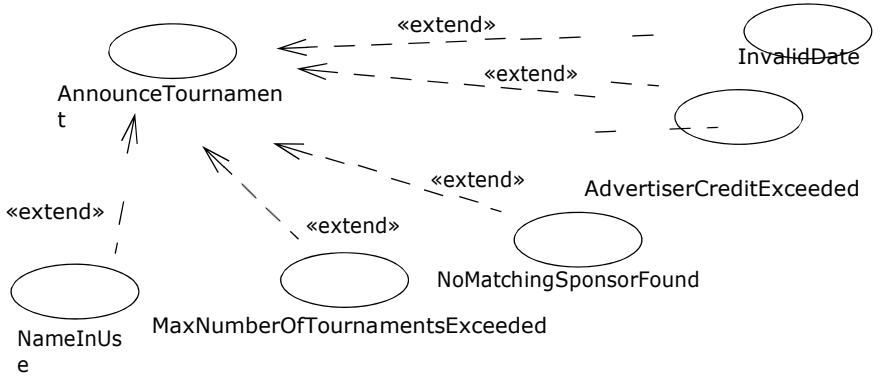
When describing the steps of the detailed AnnounceTournament use case, we and the client made more decisions about the boundaries of the system:

- We introduced start and end dates for the application process and for executing the tournament (Step 3 in Figure 4-24). This enables us to communicate deadlines to all actors involved to ensure that the tournament happens within a reasonable time frame.
- We decided that advertisers indicate in their profile whether they are interested in exclusive sponsorships or not. This enables the LeagueOwner to target Advertisers more specifically (Step 4 in Figure 4-24).
- We also decided to enable advertisers to commit to sponsorship deals through the system and automated the accounting of advertisement and the billing. This entails security and legal requirements on the system, which we document in the “quality requirements” field of the use case (Step 9 and 10 in Figure 4-24).

Note that these decisions are validated with the client. Different clients and environments can lead to evaluating trade-offs differently for the same system. For example, the decision about soliciting Advertisers and obtaining a commitment through the system results in a more complex and expensive system. An alternative would have been to solicit Advertisers via E-mail, but obtain their commitment via phone. This would have resulted in a simpler system, but more work on the part of the LeagueOwner. The client is the person who decides between such alternatives, understanding, of course, that these decisions have an impact on the cost and the delivery date of the system.

Next, we identify the exceptions that could occur during the detailed use case. This is done by reviewing every step in the use case and identifying all the events that could go wrong. We briefly describe the handling of each exception and depict the exception handling use cases as extensions of the AnnounceTournamentuse case (Figure 4-25).

Note that not all exceptions are equal, and different kinds of exceptions are best addressed at different stages of development. In Figure 4-25, we identify exceptions caused by resource constraints (MaxNumberOfTournamentsExceeded), invalid user input (InvalidDate, NameInUse), or application domain constraints (AdvertiserCreditExceeded, NoMatchingSponsorFound). Exceptions associated with resource constraints are best handled during system design. Only during system design will it become clear which resources are limited and how to best share



AdvertiserCreditExceeded The system removes the Advertiser from the list of potential sponsors.

InvalidDate The system informs the LeagueOwnerand prompts for a new date.

MaxNumberOfTournaments Exceeded The AnnounceTournamentuse case is terminated.

NameInUse The system informs the LeagueOwnerand prompts for a new name.

NoMatchingSponsorFound The system skips the exclusive sponsor steps and chooses random advertisements from the advertisement pool.

Figure 4-25 Exceptions occurring in AnnounceTournament represented as extending use cases. (Note that AnnounceTournamentin this figure is the same as the use case in Figure 4-23).

them among different users which may, in turn, trigger further requirements activities during system design to validate with the client the handling of such exceptions. Exceptions associated with invalid user input are best handled during user interface design, when developers will be able to decide at which point to check for invalid input, how to display error messages, and how to prevent invalid inputs in the first place. The third category of exceptions—application domain constraints—should receive the focus of the client and developer early. These are exceptions that are usually not obvious to the developer. When missed, they require substantial rework and changes to the system. A systematic way to elicit those exceptions is to walk through the use case step by step with the client or a domain expert.

Many exceptional events can be represented either as an exception (e.g., AdvertiserCreditExceeded) or as a nonfunctional requirement (e.g., "An Advertiser should not be able to spend more advertisement money than a fixed limit agreed beforehand with the Operator during the registration"). The latter representation is more appropriate for global constraints that apply to several use cases. Conversely, the former is more appropriate for events that can occur only in one use case (e.g., "NoMatchingSponsorFound").

Writing each detailed use case, including their exceptions, constitutes the lion's share of the requirements elicitation effort. Ideally, developers write every detailed use case and address all application domain issues before committing to the project and initiating the realization of the system. In practice, this never happens. For large systems, the developers produce a large amount of documentation in which it is difficult, if not impossible, to maintain consistency. Worse, the requirements elicitation activity of large projects should already be financed, as this phase requires a lot of resources from both the client and the development organization. Moreover, completeness at an early stage can be counterproductive: use case steps change during development as new domain facts are discovered. The decision about how many use cases to detail and how much to leave implicit is as much a question of trust as of economics: the client and the developers should share a sufficiently good understanding of the system to be ready to commit to a schedule, a budget, and a process for handling future changes (including changes in requirements, schedule, and budget).

In ARENA, we focus on specifying in detail the interactions that involve the Advertisers and the Players, since they have critical roles in generating revenue. Use cases associated with the administration of the system or the installation of new games or tournament styles are left for later, since they also include more technical issues that are dependent on the solution domain.

4.6.5 Identifying Nonfunctional Requirements

Nonfunctional requirements come from a variety of sources during the elicitation. The problem statement we started with in Figure 4-17 already specified performance and implementation requirements. When detailing the AnnounceTournament use case, we identified further legal requirements for billing Advertisers. When reviewing exceptions in the previous section, we identified a constraint on the amount of money Advertisers can spend. Although we encounter many nonfunctional requirements while writing use cases and refining them, we cannot ensure

that we identify all the essential nonfunctional requirements. To ensure completeness, we use the FURPS+ categories we described in Section 4.3.2 (or any other systematic taxonomy of nonfunctional requirements) as a checklist for asking questions of the client. Table 4-5 depicts the nonfunctional requirements we identified in ARENA after detailing the AnnounceTournament use case.

Table 4-5 Consolidated nonfunctional requirements for ARENA, after the first version of the detailed AnnounceTournament use case.

Requisiti Category	non funzionali
Usability	• Spectators must be able to access games in progress without prior registration and without prior knowledge of the Game.
Reliability	• Crashes due to software bugs in game components should interrupt at most one Tournament using the Game. The other Tournaments in progress should proceed normally. • When a Tournament is interrupted because of a crash, its LeagueOwner should be able to restart the Tournament. At most, only the last move of each interrupted Match can be lost.
Performance	• The system must support the kick-off of many parallel Tournaments (e.g., 10), each involving up to 64 Players and several hundreds of simultaneous Spectators. • Players should be able to play matches via an analog modem.
Supportability	• The Operator must be able to add new Games and new TournamentStyles. Such additions may require the system to be temporarily shut down and new modules (e.g., Java classes) to be added to the system. However, no modifications of the existing system should be required.
Implementation	• All users should be able to access an Arena with a web browser supporting cookies, Javascript, and Java applets. Administration functions used by the operator are not available through the web. • ARENA should run on any Unix operating system (e.g., MacOS X, Linux, Solaris).
Operation	• An Advertiser should not be able to spend more advertisement money than a fixed limit agreed beforehand with the Operator during the registration.
Legal	• Offers to and replies from Advertisers require secure authentication, so that agreements can be built solely on their replies. • Advertisers should be able to cancel sponsorship agreements within a fixed period, as required by local laws.

4.6.6 Lessons Learned

In this section, we developed an initial use case and analysis object model based on a problem statement provided by the client. We used scenarios and questions as elicitation tools to clarify ambiguous concepts and uncover missing information. We also elicited a number of nonfunctional requirements. We learned that

- Requirements elicitation involves constant switching between perspectives (e.g., high-level vs. detailed, client vs. developer, activity vs. entity).
- Requirements elicitation requires a substantial involvement from the client.
- Developers should not assume that they know what the client wants.
- Eliciting nonfunctional requirements forces stakeholders to make and document trade-offs.

4.7 Further Readings

The concept of use case was made popular by Ivar Jacobson in his landmark book, *Object-Oriented Software Engineering: A Use Case Approach* [Jacobson et al., 1992]. For an account of the early research on scenario-based requirements and, more generally, on participatory design, *Scenario-Based Design* [Carroll, 1995] includes many papers by leading researchers about scenarios and use cases. This book also describes limitations and pitfalls of scenario-based requirements and participatory design, which are still valid today.

For specific method guidance, *Software for Use* [Constantine & Lockwood, 1999] contains much material on specifying usable systems with use cases, including eliciting imprecise knowledge from users and clients, a soft topic that is usually not covered in software engineering text books. *Writing Effective Use Cases* [Cockburn, 2001] and its accompanying website http://www.usecases.org provide many practical heuristics for writing use cases textually (as opposed to just drawing them).

End users play a critical role during requirements elicitation. Norman illustrates this by using examples from everyday objects such as doors, stoves, and faucets [Norman, 2002]. He argues that users should not be expected to read a user manual and learn new skills for every product to which they are exposed. Instead, knowledge about the use of the product, such as hints indicating in which direction a door opens, should be embedded in its design. He takes examples from everyday objects, but the same principles are applicable to computer systems and user interface design.

The world of requirements engineering is much poorer when it comes to dealing with nonfunctional requirements. The NFR Framework, described in *Non-Functional Requirements in Software Engineering* [Chung et al., 1999], is one of the few methods that addresses this topic systematically and thoroughly.

The RAD template introduced in this chapter is just one example of how to organize a requirements document. IEEE published the documentation standard IEEE-Std 830-1998 for

software requirements specifications [IEEE Std. 830-1998]. The appendix of the standard contains several sample outlines for the description of specific requirements.

The examples in this chapter followed a dialectic approach to requirements elicitation, a process of discussion and negotiation among developers, the client, and the end users. This approach works well when the client is the end user, or when the client has a sufficiently detailed knowledge of the application domain. In large systems, such as an air traffic control system, no single user or client has a complete perspective of the system. In these situations, the dialectic approach breaks down, as much implicit knowledge about the users' activities is not encountered until too late. In the past decade, ethnography, a field method from anthropology, has gained popularity in requirements engineering. Using this approach, analysts immerse themselves in the world of users, observe their daily work, and participate in their meetings. Analysts record their observations from a neutral point of view. The goal of such an approach is to uncover implicit knowledge. The coherence method, reported in *Social analysis in the requirements engineering process: from ethnography to method* [Viller & Sommerville, 1999], provides a practical example of ethnography applied to requirements engineering.

Managing traceability beyond requirements is still a research topic, the reader is referred to the specialized literature [Jarke, 1998].

Finally, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices* [Jackson, 1995] is a concise, incisive, and entertaining piece that provides many insights into principles and methods of requirements engineering.

4.8 Exercises

4-1 Consider your watch as a system and set the time 2 minutes ahead. Write down each interaction between you and your watch as a scenario. Record all interactions, including any feedback the watch provides you.

4-2 Consider the scenario you wrote in Exercise 4-1. Identify the actor of the scenario. Next, write the corresponding use case SetTime. Include all cases, and include setting the time forward and backward, and setting hours, minutes, and seconds.

4-3 Assume the watch system you described in Exercises 4-1 and 4-2 also supports an alarm feature. Describe setting the alarm time as a self-contained use case named SetAlarmTime.

4-4 Examine the SetTime and SetAlarmTime use cases you wrote in Exercises 4-2 and 4-3. Eliminate any redundancy by using an include relationship. Justify why an include relationship is preferable to an extend relationship in this case.

4-5 Assume the FieldOfficer can invoke a Help feature when filling an EmergencyReport. The HelpReportEmergency feature provides a detailed description for each field and specifies which fields are required. Modify the ReportEmergency use case (described in Figure 4-10) to include this help functionality. Which relationship should you use to relate the ReportEmergency and HelpReportEmergency?

4-6 Below are examples of nonfunctional requirements. Specify which of these requirements are verifiable and which are not:

- "The system must be usable."
- "The system must provide visual feedback to the user within one second of issuing a command."
- "The availability of the system must be above 95 percent."
- "The user interface of the new system should be similar enough to the old system that users familiar with the old system can be easily trained to use the new system."

4-7 The need for developing a complete specification may encourage an analyst to write detailed and lengthy documents. Which competing quality of specification (see Table 4-1) may encourage an analyst to keep the specification short?

4-8 Maintaining traceability during requirements and subsequent activities is expensive, because of the additional information that must be captured and maintained. What are the benefits of traceability that outweigh this overhead? Which of those benefits are directly beneficial to the analyst?

4-9 Explain why multiple-choice questionnaires, as a primary means of extracting information from the user, are not effective for eliciting requirements.

4-10 From your point of view, describe the strengths and weaknesses of users during the requirements elicitation activity. Describe also the strengths and weaknesses of developers during the requirements elicitation activity.

4-11 Briefly define the term "menu." Write your answer on a piece of paper and put it upside down on the table together with the definitions of four other students. Compare all five definitions and discuss any substantial difference.

4-12 Write the high-level use case ManageAdvertisement initiated by the Advertiser, and write detailed use cases refining this high-level use case. Consider features that enable an Advertiser to upload advertisement banners, to associate keywords with each banner, to subscribe to notices about new tournaments in specific leagues or games, and to monitor the charges and payments made on the advertisement account. Make sure that your use cases are also consistent with the ARENA problem statement provided in Figure 4-17.

4-13 Considering the AnnounceTournament use case in Figure 4-24, write the event flow, entry conditions, and exit conditions for the use case ApplyForTournament, initiated by a Player interested in participating in the newly created tournament. Consider also the ARENA problem statement provided in Figure 4-17. Write a list of questions for the client when you encounter any alternative.

4-14 Write the event flows, entry conditions, and exit conditions for the exceptional use cases for AnnounceTournament depicted in Figure 4-25. Use include relationships if necessary to remove redundancy.

References

[Bruegge et al., 1994] B. Bruegge, K. O'Toole, & D. Rothenberger, "Design considerations for an accident management system," in M. Brodie, M. Jarke, M. Papazoglou (eds.), *Proceedings of the Second International Conference on Cooperative Information Systems*, pp. 90–100, University of Toronto Press, Toronto, Canada, May 1994.

[Carroll, 1995] J. M. Carroll (ed.), *Scenario-Based Design: Envisioning Work and Technology in System Development.* Wiley, New York, 1995.

[Chung et al., 1999] L. Chung, B. A. Nixon, E. Yu & J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic, Boston, 1999.

[Cockburn, 2001] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, Reading, MA, 2001.

[Constantine & Lockwood, 1999] L. L Constantine & L. A. D. Lockwood, *Software for Use*, Addison-Wesley, Reading, MA, 1999.

[Grady, 1992] R. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[Hammer & Champy, 1993] M. Hammer & J. Champy, *Reengineering The Corporation: a Manifesto For Business Revolution*, Harper Business, New York, 1993.

[IEEE Std. 610.12-1990] IEEE, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, New York, NY, 1990.

[IEEE Std. 830-1998] *IEEE Standard for Software Requirements Specification*, IEEE Standards Board, 1998.

[ISO Std. 9126] International Standards Organization. *Software engineering—Product quality*. ISO/IEC-9126, Geneva, Switzerland, 2001.

[Jackson, 1995] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, Reading, MA, 1995.

[Jacobson et al., 1992] I. Jacobson, M. Christerson, P. Jonsson, & G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach*, Addison-Wesley, Reading, MA, 1992.

[Jacobson et al., 1999] I. Jacobson, G. Booch, & J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[Jarke, 1998] M. Jarke, "Requirements tracing," *Communications of the ACM*, Vol. 41, No. 12, December 1998.

[Neumann, 1995] P. G. Neumann, *Computer-Related Risks*, Addison-Wesley, Reading, MA, 1995. [Nielsen, 1993]

J. Nielsen, *Usability Engineering*, Academic, New York, 1993.

[Norman, 2002] D. A. Norman, *The Design of Everyday Things*, Basic Books, New York, 2002.

[Rational] Rationale, http://www.rational.com.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Telelogic] Telelogic, http://www.telelogic.se.

[Viller & Sommerville, 1999] S. Viller & I. Sommerville, "Social analysis in the requirements engineering process: from ethnography to method," *International Symposium on Requirements Engineering (ISRE'99)*, Limerick, Ireland, June 1999.

[Wirfs-Brock et al., 1990] R. Wirfs-Brock, B. Wilkerson, & L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Wood & Silver, 1989] J. Wood & D. Silver, *Joint Application Design*®, Wiley, New York, 1989.