POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

SAMA: Un framework per applicazioni mobili adattive

Relatore: Prof. Carlo GHEZZI

Correlatore: Ing. Giordano TAMBURRELLI

Tesi di Laurea di:

Tengda Huang

Matricola n. 771273

Anno Accademico 2012-2013

Alla mia famiglia
e a tutte le persone
che mi sono stati
vicini in questi anni

Ringraziamenti

Desidero ringraziare tutti coloro che in questi anni mi sono stati vicini, accompagnandomi nel percorso che mi ha fatto giungere a questo importante traguardo.
Un grazie particolare spetta ovviamente alla mia famiglia, che mi ha sempre aiutato e supportato nelle mie scelte. Gran parte di questo risultato è merito vostro.
Grazie anche ai miei amici di sempre, che mi sono stati vicini anche quando il tempo per vederci non è stato molto.
Un grazie molto sentito anche per i miei compagni di corso, che sono riusciti a rendere questi due anni davvero indimenticabili.
Infine vorrei ringraziare chi mi ha seguito e chi ha contribuito a questo lavoro, facendo in modo che fosse sempre interessante e stimolante.

Tengda

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Abstract

Recentemente vi è stata una massiccia diffusione dei dispositivi mobili che ha comportato una sempre più grande richiesta di applicazioni mobili. A differenza dei sistemi software tradizionali, lo sviluppo di applicazioni mobili è caratterizzato da una maggiore, spesso esplicita, dipendenza dalle caratteristiche hardware e software del dispositivo in cui l'applicazione viene utilizzata. Infatti, le applicazioni possono essere installate su dispositivi anche molto diversi tra loro. Inoltre, la strategia di sviluppo delle applicazioni mobili spesso si basa sul riutilizzare servizi e/o applicazioni esterne, e questo introduce una dipendenza anche verso questi ultimi. Per far fronte a queste peculiarità, le applicazioni devono essere adattive sia rispetto agli ambienti di esecuzione sia per quanto riguarda i servizi e le applicazioni esterne su cui fanno affidamento.
L'approccio tradizionale consiste nel definire in modo imperativo le strategie di adattamento utilizzando contorti flussi di esecuzione, e complessi strategie di controllo per la gestione degli errori. Questo ha come risultato un codice complesso che intreccia la logica dell'applicazione con la logica di adattamento, rendendo il codice poco leggibile e facilmente soggetto ad errori.
In questa tesi viene proposto un'approccio, chiamato *SAMA*, basato sui principi del *Dynamic Software Product Lines*. In particolare, il framework

separa la logica di adattamento dalla logica applicativa, utilizzando un *feature model* per la modellazione della variabilità, e affidando ad un componente middleware la sua gestione.
I vantaggi dell'approccio sono stati mostrati attraverso un esempio di applicazione ispirato a diverse applicazioni esistenti.

Capitolo 1

Introduzione

Recentemente vi è stata una massiccia diffusione dei dispositivi mobili, come i vari smartphone e tablet, arrivando a supportare le persone nelle loro attività quotidiane. La crescita di questo fenomeno è guidata dalle *apps*, che possono essere facilmente scaricate ed installate sul dispositivo. Ogni settimana vengono pubblicate migliaia di nuove applicazioni in tutto il mondo. A differenza dei sistemi software tradizionali, lo sviluppo di applicazioni mobili però è caratterizzato da una maggiore, spesso esplicita, dipendenza dalle caratteristiche hardware e software del dispositivo in cui le applicazioni vengono installate. Infatti, anche se è sviluppata per una specifica piattaforma (ad esempio, iOS o Android), la stessa applicazione può essere installata su un insieme di dispositivi diversi caratterizzati da configurazioni eterogenee sia dal punto di vista hardware che software (e.g., sensori e hardware di rete, lista di componenti pre-installati, versione OS, ecc.). Inoltre, per rispettare i forti vincoli di time-to-market imposti dal mercato di applicazioni mobile, spesso le applicazioni vengono sviluppate integrando componenti e funzionalità spesso riutilizzati da altre applicazioni. Questo introduce una dipendenza anche verso componenti esterni

(e.g., servizi disponibili online) che possono non essere sempre disponibili. Lo sviluppatore quindi deve ricordare che il contesto può cambiare a tempo di esecuzione e può sovvertire le ipotesi fatte a tempo di progettazione, compromettendo il corretto funzionamento dell'applicazione. Come esempio, si consideri il caso di un'applicazione iPhone che utilizza la fotocamera integrata. L'iPhone attuale ha una fotocamera auto focus mentre le versioni precedenti, ancora in uso diffuso, sono equipaggiati da fotocamera fixed focus. Questa differenza, seppur apparentemente poco significativa, se non gestita può influenzare la capacità dell'applicazione di soddisfare i suoi requisiti. Per far fronte a queste peculiarità, le applicazioni devono essere *adattive* [1, 2] sia rispetto agli ambienti di esecuzione eterogenei sia per quanto riguarda i servizi e le applicazioni esterne su cui fanno affidamento.

Un approccio tradizionale per implementazione di questo comportamento adattivo è quello di programmare esplicitamente gli adattamenti necessari a livello statico, utilizzando flussi di esecuzione composti da intricati rami di if-else, e di tecniche di gestione delle eccezioni per gli scenari inaspettati che si verificano. Questo ha come risultato un codice complesso che intreccia la logica dell'applicazione con la logica adattiva che gestisce le varie tipologie di dispositivi, rendendo il codice poco leggibile e facilmente soggetto ad errori.

Nella tesi viene proposto un approccio differente, il framework *SAMA* (Self-Adaptive Mobile Application). Il framework è stato progettato soprattutto per applicazioni mobili per la piattaforma Android. L'obiettivo è di rendere l'applicazione *adattiva* rispetto al contesto di esecuzione [3], cercando di separare la logica applicativa dell'applicazione dalla logica di adattamento al contesto di esecuzione. In particolare, l'adattamento viene

gestito dal framework a livello runtime attraverso un middleware.

SAMA riprende i principi delle Software Product Lines (SPL) [4], e l'estensione Dynamic Software Product Lines (DSPL) [5]. L'approccio prevede una "modellazione dello spazio di variabilità". Per spazio di variabilità s'intende le possibili variazioni del comportamento dell'applicazione a seconda dell'ambiente di esecuzione.
Il processo di modellazione consiste in un "astrazione del dominio dell'applicazione", che individua le funzionalità dell'applicazione e le organizza in modo gerarchico in una struttura ad albero chiamato *feature model* (FM). Ogni funzionalità corrisponde ad una *feature* nell'albero.
La peculiarità di questo modello è la *variabilità*, la sua struttura si adatta automaticamente al contesto. La variabilità è determinata dal fatto che le feature possono essere in uno stato *attivo*, oppure *inattivo*. Lo stato di una feature è determinato a livello runtime e può dipendere dal contesto di esecuzione. A livello runtime, le feature inattive vengono eliminate dall'albero, quindi la struttura è determinata dall'insieme delle feature attive. L'idea principale è di sfruttare questa caratteristica di variabilità del FM, così da riuscire a dare anche all'applicazione un comportamento variabile. Per fare ciò, la progettazione della struttura dell'applicazione è basata sul FM. In particolare, dal FM vengono generati i componenti che determineranno la struttura della logica applicativa dell'applicazione. Così come il FM si adatta al contesto, anche l'architettura dell'applicazione fa lo stesso. A livello runtime, la configurazione della struttura è affidata ad un middleware, che utilizza la *Dependency Injection* per la gestione dei componenti. In particolare, il middleware controlla che vengano istanziate solo componenti corrispondenti alle feature attive. Le regole della dependency injection corrisponderanno alle regole di attivazione delle feature, e sono

anch'esse generate dal framework a partire dal FM.

Come caso studio per mostrare il funzionamento di SAMA è stata utilizzata l'applicazione *Winery*, un'applicazione semplice riguardante i vini. Brevemente, Winery offre una funzionalità di ricerca dei vini, la ricerca può essere fatta a partire da un nome, oppure da un codice a barre. La funzionalità di ricerca da codice a barre è disponibile solo se la fotocamera è auto-focus. Una volta ottenuto le informazioni riguardante il vino, l'applicazione offre una funzionalità di condivisione su social network. Più precisamente, Winery offre la possibilità di condividere i vini di interesse su oguno dei social network supportati da altre applicazioni client installate sul dispositivo.
Un esempio di comportamento adattivo in Winery è la funzionalità di ricerca da codice a barre, che è presente o meno a seconda delle caratteristiche della fotocamera del dispositivo, e la funzionalità di condivisione, che dipende dalle applicazioni presenti sul dispositivo.
Nella tesi vedremo come il framework gestisce questo comportamento a livello runtime, senza che il programmatore debba inserire pezzi di codice manualmente (ad esempio, rami if-else, oppure costrutti try-catch). In particolare, l'adattamento viene gestito attraverso un cambiamento della struttura dei componenti in base al contesto.

La presente tesi è organizzata come segue. Nel capitolo 2 vengono spiegati le motivazioni che hanno portato allo sviluppo del framework SAMA, specificando la necessità di adattività di un'applicazione mobile. Il capitolo 3 contiene lo stato dell'arte riguardante i sistemi adattivi e i framework per il supporto alle applicazioni adattive. Nel capitolo 4 e' riportata la descrizione dettagliata del framework SAMA. Nel capitolo 5 è descritto un caso di studio con un esempio di applicazione realizzata con il

framework SAMA. Infine, il capitolo 6 presenta le conclusioni di questo lavoro e spiega i possibili futuri sviluppi.

Capitolo 2

Motivazioni

Il software è ormai un elemento molto importante per la società moderna. Al giorno d'oggi, la maggior parte delle attività umane coinvolgono software oppure sono completamente gestite dal software. La recente diffusione massiccia dei dispositivi mobili, come i vari smartphone e tablet, che supportano le persone nelle loro attività quotidiane, rende questo fenomeno ancora più rilevante. I dispositivi mobili hanno fatto diventare i software letteralmente onnipresenti.

Inventate da Apple per il suo sistema operativo iOS e successivamente adottate da Google per il sistema operativo Android, le *apps* stanno guidando la crescita di questo fenomeno mobile. Esse di solito sono di piccole dimensioni, spesso distribuite o di tipo single-task, che l'utente può facilmente scaricare (anche gratuitamente) ed installare sul suo dispositivo per aggiungere nuove funzionalità rispetto a quelle che vengono pre-installati.

Il mercato dei dispositivi mobili che permette questa interazione è un ecosistema estremamente dinamico e vivace caratterizzato da migliaia di nuove applicazioni pubblicate in tutto il mondo ogni settimana. Questo fenomeno sta proponendo nuove sfide per l'ingegneria del software mo-

derna, prima di tutto la necessità di strategie di sviluppo efficaci incentrate su forti vincoli di tempi di mercato. Per rispondere a questa sfida di solito viene adottato un processo di sviluppo basato su componenti. Ciò è reso possibile dagli stessi framework di sviluppo creati dai moderni sistemi operativi mobili, che permettono ai componenti installati sullo stesso dispositivo di comunicare ed invocare l'un l'altro. Di conseguenza, molte applicazioni mobili sono state sviluppate mettendo insieme: (1) componenti sviluppati ad-hoc, (2) servizi esistenti disponibili on-line, (3) applicazioni di terze parti, e (4) componenti dipendenti dalla piattaforma per accedere specifiche funzionalità hardware del dispositivo (ad esempio, la camera fotografica, GPS, ecc.).

L'approccio tipico per sviluppare tali software eterogenei segue un approccio (tipicamente iterativo) a tre fasi:

- Gli sviluppatori prima individuano la lista delle funzionalità necessarie e le organizzano in un flusso di esecuzione appropriato;
- Valutano il trade-off tra implementare direttamente tali funzionalità o ricorrere a servizi esistenti o ad applicazioni di terze parti;
- Costruiscono l'applicazione implementando i componenti identificati e integrando tutti i pezzi insieme.

Costruire applicazioni come composizione di componenti, servizi e/o altre applicazioni di terze parti, tuttavia, introduce una dipendenza diretta dell'applicazione finale rispetto ai componenti esterni che a loro volta possono guastarsi, o essere indisponibili nel corso del tempo, compromettendo così le funzionalità dell'applicazione. Inoltre, a differenza dei sistemi software tradizionali, lo sviluppo di applicazioni mobili è caratterizzato da una maggiore, spesso esplicita, dipendenza dalle caratteristiche

hardware e software del dispositivo in cui viene utilizzato. Infatti la stessa applicazione, anche se è sviluppato per una specifica piattaforma (ad esempio, iOS o Android), può essere installata su un insieme di dispositivi diversi (Figura 2.1) caratterizzati da configurazioni hardware e software eterogenee (es., sensori presenti e hardware di rete, lista di componenti pre-installati, versione del sistema operativo, ecc.). Si consideri ad esempio il caso di un'applicazione iPhone che utilizza la fotocamera integrata. L' iPhone attuale ha una fotocamera auto-focus mentre le versioni precedenti, ancora in uso diffuso, sono equipaggiate da fotocamere fixed-focus. Questa differenza, seppur apparentemente poco significativa, se non gestita, può influenzare la capacità dell'applicazione di soddisfare requisiti previsti.

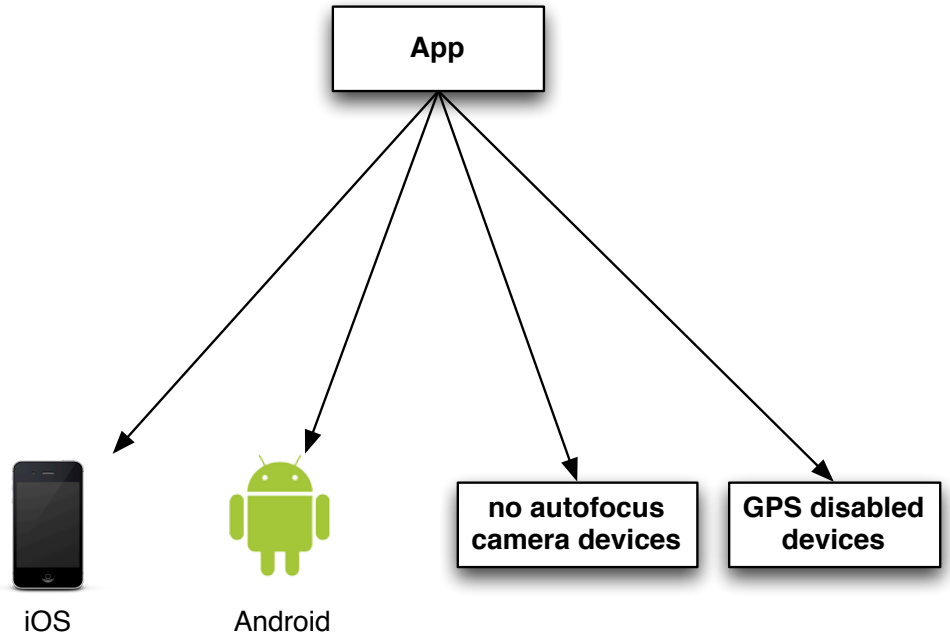

Figura 2.1: Eterogeneità dei dispositivi

Per far fronte a queste peculiarità, le applicazioni devono essere *adattive* [1, 2] sia rispetto agli ambienti di esecuzione eterogenei sia per quanto riguarda i servizi esterni e le applicazioni esterne su cui fanno affidamento. Il modo tradizionale per raggiungere questo obiettivo è quello di programmare esplicitamente gli adattamenti necessari attraverso contorti flussi di esecuzione ed attraverso tecniche di gestione di eccezioni per gli scenari inaspettati quando si verificano. Questo non è facile da realizzare e si traduce in un codice complesso che intreccia la logica dell'applicazione con la logica adattiva che gestisce le peculiarità delle varie tipologie di dispositivi e le situazioni inaspettate che possono accadere a tempo di esecuzione. Questo aggiunge una complessità ulteriore all'applicazione che risulta in un codice difficile da leggere e da mantenere. Questo lavoro di tesi si occupa nello specifico di questo problema e supporta lo sviluppo di applicazioni auto adattive.

Prendiamo come esempio *ShopSavvy*, un'applicazione esistente disponibile per la piattaforma Android. ShopSavvy permette agli utenti di condividere varie informazioni riguardanti un prodotto commerciale. In particolare, un utente di ShopSavvy può usare l'applicazione per pubblicare il prezzo di un prodotto in un certo negozio (scelto tra quelli vicini alla sua attuale posizione). In risposta, l'applicazione mostra all'utente dei posti vicini alternativi dove lo stesso prodotto è venduto ad un prezzo più conveniente. Il mapping tra il prezzo segnalato dall'utente e il prodotto è ottenuto sfruttando il suo codice a barre. In più, gli utenti possono condividere le loro opinioni riguardante il negozio dove hanno comprato il prodotto e il suo prezzo su social network, come Twitter.

Come già accennato, il processo di sviluppo di un'applicazione di questo genere parte dalla lista delle funzionalità necessarie, per poi cercare di

distinguere quelle che devono essere implementate con componenti ad-hoc e quelle che possono essere realizzate utilizzando soluzioni già esistenti (servizi esterni disponibili online oppure applicazioni che sono già installati sul dispositivo).

Nel fare queste scelte lo sviluppatore deve ricordare che il contesto a tempo di esecuzione può cambiare e può sovvertire le ipotesi fatte a tempo di progettazione, compromettendo il corretto funzionamento dell'applicazione. Per esempio, gli sviluppatori devono considerare le differenze dei vari dispositivi dove l'applicazione verrà installata, e cercare di fare in modo che l'applicazione si *adatti* ai diversi dispositivi. Analogamente, devono prendere le giuste scelte per minimizzare l'impatto dei cambiamenti dei servizi esterni utilizzati dall'applicazione, facendo in modo tale che l'applicazione si adatti a questi cambiamenti, oppure scegliere di non utilizzarli più, con il risultato però di dover implementare una funzionalità che può essere facilmente trovata online.

Date queste premesse, supponiamo di voler implementare la funzionalità di Shopsavvy della ricerca del codice a barre come un componente sviluppato ad-hoc. Questa funzionalità richiede l'utilizzo della fotocamera per estrarre il numero del codice a barre dall'immagine che l'utente acquisisce nella fotocamera. Più precisamente la fotocamera deve essere di tipo auto focus (cioè la focalizzazione avviene automaticamente) affinché funzioni correttamente. Ma come abbiamo detto l'applicazione può essere installata su dispositivi anche molto diversi tra loro, quindi può succedere che il dispositivo non abbia una fotocamera di tipo auto focus, in tal caso l'applicazione potrebbe non funzionare correttamente. Per far fronte a questa situazione e permettere un corretto riconoscimento del codice a barre anche sui dispositivi con fotocamere fixed-focus, l'applicazione deve in

qualche modo avere qualche forma di *adattività*. Infatti, deve rilevare se la fotocamera del dispositivo supporta auto focus; nel caso in cui non lo supporta, deve evocare un servizio esterno per processare l'immagine acquisita. Un'approccio simile può essere usato anche per ottenere la posizione dell'utente, che in principio richiederebbe un GPS, un componente hardware che potrebbe però non essere disponibile su tutti i dispositivi. Quindi un'implementazione alternativa è quella di mostrare una mappa all'utente affinché inserisca un'indicazione manuale della posizione attuale.

Il pezzo di codice riportato nel listato 2.1 descrive una possibile implementazione per la piattaforma Android del comportamento adattivo descritto in precedenza ([6]). Come possiamo vedere, in questo pezzo di codice il comportamento adattivo è implementato attraverso l'utilizzo di istruzioni a cascata di*if-else*. Sebbene questo sia solo un piccolo frammento di codice, si può notare come l'utilizzo di rami if-else per la gestione dei possibili casi alternativi che possono verificarsi, renda il codice contorto e facilmente soggetto a errori. Inoltre, a livello statico è molto difficile riuscire a prevedere tutti i casi possibili, quindi durante l'esecuzione può capitare che dei casi non siano stati gestiti. La situazione diventa ancora più complessa se prendiamo in considerazione le eccezioni che possono verificarsi a tempo di esecuzione, per esempio un errore durante l'accesso al GPS oppure invocando un servizio esterno, i quali devono essere gestiti dal codice in maniera esplicita, quindi eventualmente con l'aggiunta di altri rami if-else o costrutti try-catch.

La ragione principale alla base di tali problemi è che le piattaforme principali per lo sviluppo di applicazioni mobili sono basati su tradizionali linguaggi imperativi, dove il flusso di esecuzione deve essere programmato in modo esplicito. Il codice adattivo, rappresentato nel listato

2.1, è intrecciato con la logica dell'applicazione, riducendo la leggibilità e la manutenzione del codice, e ostacolando la sua futura evoluzione in termini di aggiunta di caratteristiche nuove o alternative, che richiedono rami supplementari da aggiungere.

```
1  PackageManager mng = getPackageManager ( ) ;
2  if (mng.hasSystemFeature(PackageManager.FEATURE_CAMERA_AUTOFOCUS)){
3          //Run local barcode recognition
4  }else{
5          / / Invoke remote service with blurry decoder algorithm
6
7  Location location = null ;
8  if(mng.hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS)){
9          LocationProvider provider = LocationManager .GPS PROVIDER;
10         LocationManager locManager =
11                 (LocationManager) getSystemService(Context.LOCATION SERVICE);
12         try{
13                 // Return null if the GPS signal is currently not available
14                 location = locManager.getLastKnownLocation(provider );
15         } catch ( Exception e ) {
16                 location = null ;
17         }
18 }
19
20 if ( location==null ){
21         // Device without GPS or an exception was raised invoking it
22         //We show up a map to allow the user to indicate
23         // its location manually
24         showMap ( ) ;
25 }
```

Listing 2.1: Esempio di codice adattivo

Si noti che questi concetti si applicano anche nei casi in cui si evocano applicazioni di terze parti, per esempio le applicazioni per la condivisione sui social network. Queste applicazioni di solito sono installate di default sui dispositivi, ma possono essere rimosse dall'utente, compromettendo così la capacità dell'applicazione di svolgere i suoi compiti. Per applicazio-

ni di terze parti intendiamo altre applicazioni installate sul dispositivo invocate dalla nostra applicazione per ottenere delle funzionalità specifiche, in modo da non sviluppare un componente ad-hoc per tale funzionalità.

Un esempio di applicazione di terze parti per social network è *twicca* [7]. Twicca è un'applicazione che funge da client Twitter per la piattaforma Android. Con questa applicazione l'utente non è più costretto ad aprire il browser per utilizzare Twitter.

Tornando all'esempio di ShopSavvy, abbiamo detto che offre anche una funzionalità di condivisione su social network, quindi Twitter. Per implementare questa funzionalità, l'applicazione può invocare ad esempio twicca. Un'esempio di frammento di codice per invocare twicca da altre applicazioni è rappresentato nel listato 2.2. Questo frammento di codice è stato tratto dal sito web di twicca, questo per testimoniare come al giorno d'oggi sia comune l'utilizzo di queste applicazioni di terze parti, che ormai nel web si possono trovare frammenti di codice da utilizzare nell'applicazione per evocarli.

Riassumendo, abbiamo visto come l'esecuzione di un'applicazione mobile possa essere influenzato sia da caratteristiche hardware e software del dispositivo, sia da componenti esterni. Per questo è molto importante che un'applicazione mobile sia adattivo. Inoltre abbiamo visto che la gestione dell'adattività a livello statico, per esempio attraverso rami nel flusso di esecuzione, risulta in un codice complesso e facilmente soggetto ad errori e di conseguenza difficile da leggere e da mantenere. In questa tesi proporremo un nuovo approccio, chiamato "SAMA" (Self Adaptive Mobile Application), dove l'adattività è gestita automaticamente dal framework attraverso un componente middleware, riuscendo così a non intrecciare la logica applicativa con la logica di adattamento.

```
1  public static void tweet( Context context, String tweet ){
2          Uri.Builder builder = Uri.parse( "https://twitter.com/intent/tweet"
               ).buildUpon();
3          builder.appendQueryParameter( "text", tweet ); // WebIntent parameter.
4          // Add WebIntent parameter here if you need.
5
6          Uri web_intent_uri = builder.build();
7
8          try{
9                  // Try to open tweet via twicca.
10                 Intent intent = new Intent( Intent.ACTION_VIEW, web_intent_uri );
11                 intent.addCategory( Intent.CATEGORY_DEFAULT );
12                 intent.setPackage( "jp.r246.twicca" );
13                 context.startActivity( intent );
14                 return;
15         }
16         catch( ActivityNotFoundException e ){
17                 // twicca is not intalled.
18
19                 // If you need, you can customize here for your own
                       authentication.
20
21                 // Open Twitter's WebIntents in browser.
22                 Intent intent = new Intent( Intent.ACTION_VIEW, web_intent_uri );
23                 intent.addCategory( Intent.CATEGORY_DEFAULT );
24                 context.startActivity( intent );
25                 return;
26         }
27 }
```

Listing 2.2: Frammento di codice per evocare twicca

Capitolo 3

Stato dell'arte

Recentemente, molte ricerche sono state focalizzate sullo sviluppo efficace ed efficiente di applicazioni mobili, come possiamo vedere in [8] e [9]. In questo capitolo presentiamo i precedenti lavori e/o studi da cui questa tesi ha preso spunto.

Abbiamo diviso i lavori correlati in tre categorie. Inanzittutto sono stati analizzati i lavori riguardanti i *sistemi auto-adattivi*, cercando di trarre aspetti interessanti riguardante il legame tra il contesto di esecuzione e il comportamento dei sistemi software. Poi abbiamo esaminato i lavori riguardante le *Software Product Lines* su cui è basato il framework SAMA. Infine i *feature model*, che sono un elemento fondamento del framework per la modellazione della struttura dell'applicazione.

Il capitolo quindi è strutturato nella seguente maniera: la sezione 3.1 descrive i lavori sui sistemi auto-adattivi; sezione 3.2 riguarda Software Product Lines; infine la sezione 3.3 sui feature model.

3.1 Sistemi auto-adattivi

3.1.1 Context-aware

Il framework oggetto della tesi ha come obiettivo quello di rendere adattiva l'applicazione mobile rispetto al contesto in cui viene eseguito. Quindi Inanzitutto introduciamo i sistemi auto-adattivi che si basano su framework *context-aware*. Gli approcci dei framework context-aware consistono nello sviluppo delle applicazioni mobile che sono sensibili al contesto di distribuzione (e.g., la piattaforma hardware) e al contesto di esecuzione (e.g., la posizione dell'utente) [3]. Per esempio Subjective-C, [10] per supportare applicazioni context aware scritte in Objective-C, il tradizionale linguaggio usato per la programmazione delle applicazioni iOS. Il middleware EgoSpaces [11] può essere usato per ottenere informazioni sul contesto alle applicazioni. Un altro approccio per il middleware di sviluppo mobile è presentato nel [12], il quale sfrutta i principi della reflection per supportare le funzionalità mobili adattive e dipendenti dal contesto. In generale, questi approcci supportano gli sviluppatori con delle astrazioni per interrogare il contesto attuale e rilevare cambiamenti del contesto. Nella stessa direzione, approcci come [13] e [14] forniscono delle specifiche estensioni per la piattaforma android.

Dal nostro punto di vista, gli approcci sopra menzionati non sono direttamente legati al nostro framework, piuttosto possono essere visti come approcci ortogonali. SAMA può beneficiare dalla loro abilità di rilevare informazioni riguardanti in contesto. Il valore aggiunto di SAMA è invece la sua capacità di costruire un flusso di esecuzione basato sul contesto.

3.1.2 Tecniche di adattamento

Per quanto riguarda le tecniche di adattamento, possiamo classificarle in due famiglie. Il primo è quello più utilizzato ed è basato sulle regole di tipo event-guard-action [15]. In questi approcci il contesto e le configurazioni sono correlati da un insieme di regole, che esprimono come l'evoluzione del contesto dovrebbe influenzare la configurazione in esecuzione dell'applicazione. La seconda famiglia approcci si basa sull'ottimizzazione di alcune funzioni di utilità [16] associate al sistema. In questi approcci, cambiamenti dell'ambiente innesca un processo di ottimizzazione che valuta le possibile configurazioni alternative e adatta il sistema per massimizzare l'utilità delle configurazioni in esecuzione.

3.1.3 Multi-piattaforme

Altri approcci correlati (es., [17]) forniscono soluzioni per sviluppo delle applicazioni multi-piattafome. Approcci come [18] e [19] permettono agli programmatori di sviluppare utilizzando tecnologie standard (es., Javascript e HTML5) e distribuisce lo stesso codice su diverse piattaforme, come iOS e android. Questi framework hanno una grande potenzialità, ma allo stesso tempo soffrono delle stesse limitazioni per lo sviluppo delle tradizionali app, come la logica applicativa intrecciata con codice di adattamento e un supporto limitato per la manutenzione del codice.

3.1.4 Applicazioni mobili service-oriented

Nessuno degli approcci sopra citati si occupa delle applicazioni mobile service-oriented, le quali rappresentano invece una quota significativa delle applicazioni sviluppate finora. Il lavoro discusso in [20] descrive un

approccio per la composizione dei servizi nell'ambiente mobile e valuta i criteri di valutazione dei protocolli che abilitano questa composizione. Si concentrano principalmente su un'architettura distribuita che facilita la composizione dei servizi e non si concentrano sull'application layer e sulle sue capacità di adattamento, come invece avviene in SAMA. In generale, gli approcci esistenti per le applicazioni service-oriented in ambienti mobilesi si focalizzano sull'attivazione della composizione dei servizi, senza considerare le conseguenze associate, come per esempio la necessità di adattamento descritto nel capitolo 2.

3.1.5 Approccio dichiarativo: SelfMotion

Un approccio strettamente correlato alla presente tesi e' SelfMotion [21]. A differenza di SAMA, SelfMotion utilizza un approccio dichiarativo, che permette alle applicazioni di essere modellate descrivendo: un insieme di azioni astratte, che forniscono una descrizione ad alto livello delle attività elementari che realizzano le funzionalità richieste dall'applicazione; una serie di azioni concrete, che mappano le azioni astratte alle tappe effettive da compiere per ottenere il comportamento previsto (ad esempio, invocare un servizio esterno o un'applicazione di terze parti pre-installato); un profilo QoS per ogni azione concreta che modella le sue caratteristiche non funzionali (ad esempio, il consumo di energia e larghezza di banda); l'obiettivo generale e la politica di QoS da adottare per raggiungere tale obiettivo (ad esempio, la riduzione al minimo del consumo di energia). Le applicazioni vengono poi eseguite da un middleware che sfrutta tecniche di pianificazione automatica per l'elaborare, in fase di esecuzione, la migliore sequenza di azioni astratte per raggiungere l'obiettivo, mappandole poi alle azioni concrete da eseguire secondo la politica

di QoS specificata. Ogni volta che avviene un cambiamento nell'ambiente esterno (ad esempio, un servizio diventa indisponibile), che impedisce il completamento del piano di esecuzione definito, il middleware costruisce, in modo automatico e trasparente all'utente, un piano alternativo e continua ad eseguire l'applicazione.

3.2 Software Product Lines

Il framework SAMA modelle le applicazioni mobili come *Software Product lines*. Una definizione di Sofwatre product line si può trovare in [20]: "Una Software Product Line è un insieme di sistemi software che condividono un insieme di caratteristiche comuni e gestiti, che soddisfano le esigenze specifiche di un particolare segmento di mercato e che sono stati sviluppati partendo da un insieme di elementi centrali in un modo prescritto". In questo documento vengono spiegati i principi del paradigma SPL, i suoi vantaggi, e una metodologia di utilizzo.

Un primo esempio di sistema auto-adattivo basato sul paradigma SPL si può trovare in [22]. Quest'approccio consiste nella modellazione di due dimensioni dinamici di variabilità: variabilità ambientale, che definisce le condizioni sotto cui il sistema deve adattarsi, e la variabilità strutturale che determina le configurazioni architetturali risultanti. Per la modellazione della variabiltà viene utilzzato un linguaggio DSML(domain-specific modelling language). Dai modelli vengono generati in XML il codice e le descrizioni delle configurazioni dei componenti. Le configurazioni sono determinate a tempo di esecuzione da una piattaforma middleware che utilizza delle regole di riconfigurazione, anche queste sono modellate dal linguaggio e quindi generate automaticamente a partire dai modelli. Un

altro approccio simile lo possiamo trovare in [23], dove vengono utilizzati tre livelli di dimensione: funzionale, piattaforma, topologico. La dimensione funzionale consiste in un'astrazione delle funzionalità del sistema, e quindi modella la variabilità in termini di funzionalità del sistema. Una volta definite le funzionalità, il livello piattaforma definisce le relazioni tra le funzionalità ed i componenti del sistema. Il livello topologico definisce l'insieme delle possibili configurazioni del sistema, per ogni configurazione specifica l'insieme dei componenti e le relazioni tra essi. Questi framework hanno molto in comune con SAMA, come ad esempio la definizione della variabilità mediante feature model e la generazione del codice.

In [24] troviamo un'approccio component-based, il framework in question è MADAM. L'approccio si basa sul concetto che le applicazioni adattive sono costruite come famiglie di sistemi basati su componenti, in cui in ogni famiglia sono presenti dei componenti che modellano esplicitamente la variabilità. Questi componenti monitorano il contesto durante tutta l'esecuzione dell'applicazione, e in caso di cambiamenti del contesto, riconfigura la famiglia dei componenti. MADAM utilizza annotazioni di proprietà per descrivere la loro qualità di servizio. Per esempio, un componente video streaming può avere le proprietà come tempo di startup, perdite di frame, etc. A runtime, l'adattamento è eseguito utilizzando queste proprietà e una funzione di utilità per selezionare i componenti che meglio si adattano al contesto corrente. Rispetto a SAMA, l'approccio MADAM non prende in considerazione l'impatto delle condizioni di contesto simultanea. Un'approccio molto simile a MADAM è descritto in [25], dove il paradigma SPL viene integrato mediante una architettura a plug-in. L'approccio si basa sul concetto che SPL e l'architettura a plug-in perse-

guono obiettivi complementari, SPL cerca di modellare la variabilità dei sistemi software su diversi livelli di astrazione, mentre un sistema a plug-in supporta estesibilità, personalizzazione, ed evoluzione del software. In particolare il ruolo di riconfigurazione del sistema a livello runtime è affidato ad una piattaforma plug-in, attraverso caricamento e scaricamento dei plug-ins, mentre la modellazione della variabilità è gestita da SPL.

In [26] viene proposto un'approccio globale per la progettazione di prodotti riconfigurabili dinamicamente. In particolare viene introdotto in concetto di *binding unit*, un insieme di funzioni che vengono utilizzate per identificare i componenti dell'architettura. Inoltre descrive anche diverse linee guide per costruire architetture riconfigurabili dinamicamente, alcuni suggerimenti sulla modellazione dell'ambiente (contesto) e considerazioni su come un oggetto configuratore dovrebbe funzionare. In [27] viene fornito una visione olistica della progettazione di Dynamic Software Product Lines (DSPL). In particolare in questo lavoro la variabilità viene definita direttamente nella architettura di riferimento mediante un profilo UML dedicato.Questa architettura comprende componenti che realizzano tipi di componenti (punti di variazione) attraverso piani (varianti) modellando un particolare scenario di riconfigurazione. La riconfigurazione è modellato attraverso la composizione di piani. Il lavoro descritto in [28] si incentra sulla gestione di variabilità nei processi a livello business. In particolare, si modella come un processo di evoluzione (che denota una riconfigurazione del sistema) rispetto a vincoli temporali.

Una metodologia di implementazione delle SPL sfrutta i principi della programmazione aspect-oriented. Un esempio lo troviamo in [29], dove vengono combinati diversi concetti come dynamic aspects, modelli runtime degli aspetti, rilevamento e risoluzione delle interazioni tra aspetti.

Un altro esempio è [30], con il framework K@rt. Il framework è composto da tre parti: un metamodello generico ed estensibile per la descrizione di sistemi in esecuzione ad un alto livello di astrazione; un insieme di meta-aspetti che estende il generico metamodello con il controllo dei vincoli, supervisione e connessioni con le piattaforme di esecuzione; alcune connessioni causali specifiche della piattaforma che permettono di sorvegliare i sistemi in esecuzione su diverse piattaforme di esecuzione.

3.3 Feature Model

Nella sezione 3.2 abbiamo visto una lista di approcci basati sulle SPL. Si può notare che praticamente tutti i framework prevedono la modellazione delle variabilità. Questo vale anche per SAMA, che utilizza il *feature model*.

In [31] possiamo trovare delle linee guida generali per la modellazione della variabilità attraverso i feature model. In particolare, le linee guida prevedono inanzittutto di analizzare il dominio, poi individuare le feature astraendo le conoscenze ottenute dall'analisi del dominio, e infine organizzare le feature in una struttura ad albero.

La maggior parte delle metodologie di modellazione con l'utilizzo dei feature model ha come riferimento FODA (Feature-Oriented Domain Analysis) [32]. FODA descrive una metodologia per scoprire e rappresentare le parti comuni tra sistemi software correlati. La metodologia consiste nell'identificazione delle caratteristiche più importanti e distintive di un sistema software in un dominio. Queste caratteristiche (feature) sono aspetti visibili del dominio. Le feature definiscono sia aspetti comuni del dominio sia le differenze nei sistemi correlati con utilizzo di una notazione grafica dove il dominio viene rappresentato in termini di caratteristiche obbligato-

rie, opzionali, e alternative. Sono stati fatti anche molti tentativi per estendere FODA, per esempio ODM (Organization Domain Modeling) [33] che generalizza la nozione di feature e fornisce linee guide integrali per la modellazione del dominio. L'obiettivo principale di ODM risiede nella definizione di un processo di modellazione di un dominio base, il quale è indipendente da una specifica rappresentazione del modello. FeatuRSEB [34], che estende RSEB(Reuse-Driven Software Engineering Business) [35], è un metodo di ingegneria del software orientato ad oggetti basato su notazioni UML, con il modello delle feature di FODA. Il feature model di FeatuRSEB è usato come un catalogo dell'indice delle parti comuni e variabili catturato nei modelli RSEB (es., casi d'uso e modelli degli oggetti). Gli autori di FODA hanno anche lo hanno anche esteso per affrontare le problematiche sullo sviluppo di architetture di riferimento [36], sviluppo dei componenti orientato ad oggetti [37], e per lo sviluppo specifico di software product lines [38]. La nozione di feature in SAMA riprende la definizione descritta in FODA, in cui una feature puo esssere un requisito, una funzionalità tecnica, un gruppo di funzioni, una caratteristica non funzionale, etc.

In [39] viene adottato un approccio dove viene specificata non solo la variabilità del sistema, ma anche la variabilità dell'ambiente di esecuzione. L'idea è di modellare la struttura del sistema e l'ambiente di esecuzione in due differenti SPL per poi collegarle insieme. In pratica, vengono utilizzate due feature model differenti, le quali sono connessi tra loro attraverso dei vincoli di dipendenza, che definiscono come il sistema debba essere adattato all'ambiente. L'utilizzo dei feature model permette di rappresentare in modo omogeneo il sistema, l'ambiente, e i vincoli associati. SAMA differisce da questo approccio per l'utilizzo di un unico feature model, dove viene definita la struttura del sistema in relazione all'ambiente, quindi la

modellazione dell'ambiente è già compreso nel feature model del sistema.

In [40] SPL viene utilizzato per lo sviluppo dei sistemi pervasivi. L'approccio consiste nell'utilizzo del feature model per costruire i modelli di variazione, i quali saranno utilizzati durante esecuzione per determinare le fasi in cui è necessaria una riconfigurazione del sistema software. La riconfigurazione verrà effettuata attraverso un componente autonomo. Una validazione di questo approccio è nel [41], dove viene applicata ad un sistema Smart Home, che è una rappresentazione virtuale di una casa "intelligente" ad elevata automazione. I servizi della Smart Home devone essere configurati in risposta ai cambiamenti di condizione oppure alle attività dell'utente. Per modellare questi cambiamenti viene utilizzato un feature model, dove le feature possono una precondizione di attivazione attivazione e una precondizione di disattivazione. Le precondizioni corrispondono ad un evento. Quindi per ogni feature, ogniqualvolta che si verifica un evento corrispondente alla precondizione di attivazione allora la feature si attiva, analogamente per la precondizione di disattivazione. Apparentemente sembra che questi tipi di sistemi non hanno legame con applicazioni mobili, ma in realtà SAMA riprende da questi un concetto fondamentale che è quello del *trigger*, cioè l'attivazione delle feature mediante le precondizioni.

Molte applicazioni mobili sono progettate come applicazioni service-oriented, in [42] c'è un esempio di utilizzo dei feature model in questi sistemi, da questo approccio SAMA riprende il concetto di *generazione*. I sistemi service-oriented progettano l'applicazione organizzandolo in un insieme di componenti, l'approccio consiste quindi di modellare la variazione con il feature model, e poi generare l'architettura dei componenti a partire dal feature model. In [42] viene presentato un per costruire automaticamente

modelli di componenti da un modello di feature in base al pressuposto che una feature può essere modellato come un componente. Questo processo si focalizza ad abilitare Dynamic SPL per cambiare dinamicamente un prodotto attivando oppure disattivandole feature a runtime.

Per l'analisi dei feature model, sono stati sviluppati vari tool. Un primo esempio è FAMA (FeAture Model Analyser) [43]. FAMA è uno strumento di analisi automatizzata dei feature model che integra tre paradigmi logici e i loro rispettivi risolutori: CSP (Ja-CoP), SAT (SAT4j) e BDD (JavaBDD). Utilizzando questo strumento, è possibile eseguire diverse operazioni di ragionamento come il calcolo del numero di prodotti in un SPL, ottenere la sua lista di prodotti, filtrare i prodotti secondo un criterio, oppure individuare e spiegare gli errori. Esiste inoltre un plug-in di FAMA per l'ambiente di sviluppo Eclipse. Un altro plug-in per eclipse è FeaturePlugin [44], lo strumento supporta modellazione delle feature basato sulla cardinalità, specializzazione dei diagrammi delle feature, e configurazioni basati sui diagrammi delle feature. Poi c'è pure::variants [45], un altro plug-in si Eclipse per l'analisi delle feature model e delle sue caratteristiche.

3.3.1 Linguaggio

IL feature model è una notazione grafica, ed è difficile da analizzare per gli strumenti standard dello sviluppo dei software. Molti approcci utilizzano un linguaggio DSML (Domain-Specific Modelling Language) per rappresentare il feature model. Uno dei principali linguaggio basato su testo è TVL (Text-based Variability Language) [46, 47]. La sintassi di TVL è ispirata alla sintassi di C e quindi dovrebbe apparire intuitiva per i programmatori che hanno avuto a che fare con uno dei tanti linguaggi di

programmazione simili a C. TVL è scalabile poiché è conciso poiché integra la maggior parte dei costrutti del feature model proposti nei vent'anni dopo l'avvento di FODA. TVL è formalmente definito con una grammatica LARL ed è dotato di un'implementazione di riferimento disponibile online. TVL è il linguaggio a cui SAMA si ispira maggiormente, con la differenza che TVL permette anche di specificare attributi delle feature (nel framework SAMA, gli attributi non sono indispensabili), mentre il linguaggio del framework SAMA permette di associare precondizioni alle feature.

Un altro linguaggio è FDL (Feature Description Language)[48]. A parte TVL, è l'unico linguaggio per il quale esiste una semantica formale. A differenza di TVL, non supporta gli attributi, decomposizione basato sulla cardinalità e altri costrutti avanzati. La sintassi di FDL ha uno stile simile ad una grammatica, di cui si è ispirato il linguaggio di SAMA, infatti prevede una definizione per ogni feature, dove nella parte sinistra c'è il nome della feature, e nella parte destra i suoi figli.

Tra gli altri linguaggi troviamo Batory [49], il quale propone una sintassi GUIDSL, in cui il feature model è rappresentato da una grammatica. Il formato GUIDSL è facile da scrivere, leggere è capire. Tuttavia, non supporta un'arbitraria decomposizione delle cardinalità, attributi, o la rappresentazione del feature model come una gerarchia.

Gli strumenti SPLOT [50] e 4WhatReason [51] utilizza la sintassi e formato file SXFM. Mentre il formato utilizza XML per i metadati e la struttura generale del file, la rappresentazione del feature model è interamente basata su testo con l'obiettivo esplicito di essere leggibile. Differisce dal formato GUIDSL per il fatto che rende la struttura dell'albero esplicita attraverso (stile Python) indentazione. Supporta la decomposizione delle cardinalità

ma non gli attributi.
In [52] è stato proposto CML (Concept Modelling Language), un linguaggio prototipo che non è stato ancora definito completamente. La sua sintassi è simile a quella delle espressioni regolari.
Il framework CVM [53] supporta la modellazione della variabilità basato su testo con VSL che ha supporto per molti costrutti.
KConfig [54] è il linguaggio di configurazione del kernel di linux, è un linguaggio di descrizione dell'interfaccia di configurazione e un file KConfig può essere interpretato come un feature model. KConfig supporta strutture con inclusione di file. Supporta solo i vincoli base i quali definiscono la presenza delle feature.

Capitolo 4

Approccio SAMA

Nel capitolo 2 si è discusso dell'importanza dell'*adattività* di un'applicazione mobile. In particolare l'applicazione deve essere in grado di riconoscere autonomamente le caratteristiche importanti del contesto di esecuzione, e in base a questo gestire opportunamente il proprio comportamento. Per ottenere questo risultato, l'approccio tradizionale consisteva in una gestione a livello statico mediante i rami if-else, rendendo il codice poco leggibile e mantenibile, e facilmente soggetto ad errori.

In questo capitolo descriviamo un nuovo tipo di approccio, chiamato SAMA[1], che riprende i principi del Software Product Lines (SPL), e l'estensione Dynamic Software Product Lines (DSPL).

La struttura del capitolo è organizzata nella seguente maniera: nella sezione 4.1 viene descritto il paradigma Software Product Lines e l'estensione Dynamic Software Product Lines; la descrizione del framework SAMA è nella sezione 4.2; infine, nella sezione 4.3 vengono spiegati i vantaggi dell'approccio.

[1] *Self Adaptive Mobile Applications*

4.1 SPL e DSPL

Software Product Lines (SPL) è un paradigma creato per la fabbricazione di linee di prodotti (es. i modelli di automobili di una certa marca). Le linee di prodotti sono sicuramente diverse tra loro, però condividono delle caratteristiche comuni. Quindi per creare nuove linee di prodotti, si può riutilizzare i componenti base che sono stati progettati per creare precedenti linee di prodotti, in modo che non è più necessario ricominciare l'intero processo di creazione, ma ripartire dai componenti già esistenti, permettendo così di fare economia di scala.

Quest'idea, inizialmente utilizzata per il settore economico, fu presa a modello anche dagli ingegneri per cercare di ottenere qualcosa che è concettualmente simile all'economia di scala. L'idea principale è quella di creare un'architettura composta da un insieme di componenti base (che possono corrispondere alle funzionalità principali dell'applicazione), da questi componenti poi possono essere facilmente aggiunti nuovi elementi (componenti variabili), creando in questo modo nuove funzionalità. Ogni possibile composizione di componenti è una *configurazione*. Questo tipo di modello aumenta la variabilità e le scelte, e allo stesso tempo riduce i tempi e costi.

SPL opera a livello statico definendo le possibili configurazioni dell'applicazione, ma non fornisce soluzioni su come adattare le configurazioni a livello runtime. Per affrontare questo problema, si utilizza un'estensione di SPL chiamato Dynamic Software Product Lines (DSPL). DSPL supporta una gestione dinamica dei componenti dell'applicazione, l'applicazione cambia automaticamente la propria configurazione (aggiunta e/o rimozione di componenti) a seconda di certi criteri. Come vedremo nel corso del capitolo, per ottenere ciò viene utilizzato un *feature model*.

Il framework SAMA sfrutta proprio questo concetto del Dynamic Software Product Lines per rendere l'applicazione *adattiva*.

4.2 Descrizione dell'approccio

In questa sezione viene descritto il framework oggetto di questa tesi. Il framework in questione è SAMA, un framework per lo sviluppo di applicazioni mobili appartenenti alla piattaforma Android [55].

La caratteristica principale di SAMA è rappresentata dal fatto che l'esecuzione "adattiva" dell'applicazione è gestita in maniera automatica dal framework a livello runtime. In particolare, come vedremo nel corso del capitolo, viene utilizzato un *feature model* per modellare la variabilità del comportamento dell'applicazione in base al contesto di esecuzione. Il feature model verrà poi rappresentato da un linguaggio DSML (Domain-Specific Modelling Language). La struttura dell'applicazione sarà determinata dal feature model, quindi a partire dal feature model verranno generati i componenti per la logica applicativa dell'applicazione. A livello runtime l'esecuzione adattiva sarà gestito da un middleware, attraverso l'utilizzo della *Dependency Injection*.

In questo capitolo viene fornito innanzitutto una descrizione dell'approccio ad alto livello. Poi verranno descritte più dettagliatamente le varie fasi dello sviluppo basato sul framework, in particolare verrà descritto il feature model, il linguaggio che lo rappresenta ed il meccanismo di generazione dei componenti a partire dal feature model. Infine, viene fornita una descrizione dell'esecuzione del framework a livello runtime.

4.2.1 Descrizione ad alto livello

L'architettura concettuale del framework è mostrato nella figura 4.1.

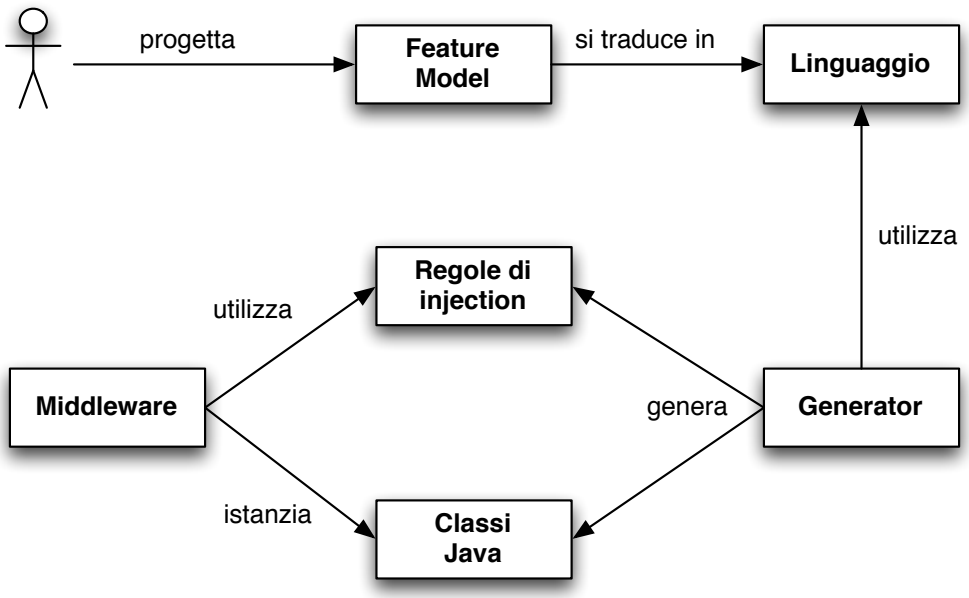

Figura 4.1: Architettura concettuale

L'obiettivo dell'approccio è di riuscire a dare all'applicazione la capacità di cambiare comportamento in base al contesto. L'applicazione non avrà un unico comportamento predeterminato, ma un insieme di possibili comportamenti.
L'approccio del framework prevede innanzitutto una "modellazione della variabilità", che consiste nella creazione di un modello che possa rappresentare l'insieme dei possibili comportamenti dell'applicazione. Per la modellazione della variabilità vengono utilizzati i principi del Dynamic Software Product Lines: ogni comportamento corrisponde ad una configurazione dei componenti (sezione 4.1), mentre una variazione del comportamento corrisponde ad un cambiamento di configurazione.

Per rappresentare le configurazioni, viene utilizzata una struttura ad albero, chiamato *feature model* (FM). L'idea è di organizzare i componenti dell'applicazione in modo gerarchico, cosicché ogni variazione può essere facilmente modellato come aggiunta e/o rimozione dei nodi dell'albero. I nodi dell'albero sono delle *feature*. In SPL una feature è una qualunque caratteristica rilevante del sistema (requisito, funzionalità tecnica, gruppo di funzioni, caratteristica non funzionale, etc.) che serve a catturare le parti comuni e variabili dei prodotti. In SAMA invece, una feature sarà considerata principalmente come una funzionalità dell'applicazione.

Il processo di costruzione del FM consiste in un "astrazione del dominio dell'applicazione": vengono individuate l'insieme delle funzionalità dell'applicazione, che vengono organizzate in modo gerarchico, per poi essere rappresentate nel FM. La feature radice quindi rappresenta l'intera esecuzione dell'applicazione. Dall'esecuzione dell'applicazione s'individuano le principali funzionalità, questi corrisponderanno alle feature figlie della radice. Poi per ogni funzionalità si cerca di capire se possono essere scomposti in altre funzionalità più specifiche, questi ultimi saranno rappresentate come feature figlie, così via fino a costruire l'intero albero.

Ogni adattamento del comportamento dell'applicazione corrisponde ad una variazione della struttura dell'albero. La capacità del FM di variare la propria struttura è determinata dal fatto che le feature hanno uno stato che può essere *attivo*, oppure *inattivo*. A livello runtime, le feature inattive vengono eliminate dall'albero, quindi la struttura è determinata solo dall'insieme delle feature attive. Lo stato di una feature è determinato a livello runtime, e dipende dal contesto di esecuzione (il criterio di determinazione dello stato, e in generale una descrizione più dettagliata del FM, è presente nella sezione 4.2.2 del capitolo). Quindi in questo modo,

la struttura del FM è determinata a livello runtime a seconda del contesto, e siccome il comportamento dell'applicazione dipende dalla struttura del FM, siamo riusciti a creare una dipendenza tra il comportamento e il contesto.

Una volta definito il feature model, il passo successivo è la sua traduzione in un linguaggio creato ad-hoc ispirato a vari linguaggi DSML (Domain-Specific Modelling Language) già esistenti [46, 47, 48].

Il FM è composto da feature, che però sono delle astrazioni delle funzionalità del sistema. Il passo successivo è la definizione dei componenti *concreti* che gestiscono l'esecuzione dell'applicazione (la logica applicativa). L'idea principale consiste nello sfruttare la capacità del FM di adattare la propria struttura in base al contesto di esecuzione. Per fare ciò, si crea una stretta corrispondenza tra la struttura del FM e la struttura della logica applicativa. In particolare l'architettura dei componenti viene generata a partire dal feature model. Quindi, così come la struttura del FM si adatta al contesto, anche la struttura della logica applicativa farà altrettanto. I componenti generati verranno in seguito implementati dallo sviluppatore a seconda delle caratteristiche della funzionalità corrispondente. L'implementazione può consistere in un' implementazione ad-hoc, utilizzo dei servizi esterni, e/o invocazione di applicazioni di terze parti.

La generazione è affidata ad un componente del framework chiamato *generator*. In particolare, il generator prende come input il linguaggio corrispondente al FM, e genera le classi Java che determineranno la logica applicativa dell'applicazione.

La gestione dei cambiamenti di configurazione, quindi struttura dell'albero, è affidata ad un middleware. Qui possiamo già notare una caratteristica molto importante di SAMA: la separazione tra logica applicativa

e la logica di adattamento.

Per la gestione della struttura il middleware utilizza la *Dependency Injection*, sfruttando la proprietà del polimorfismo della programmazione orientata agli oggetti. A livello runtime, così come il FM è determinato solo dall'insieme delle feature attive, saranno presenti solo componenti (classi Java) corrispondenti a feature attive. La creazione di oggetti Java non può più essere definito a livello statico, non si utilizza più il classico operatore "new", ma verrà effettuata appunto attraverso la Dependency Injection. Le regole di injection riprendono le regole riguardante la determinazione dello stato di una feature, e sono anch'esse generate dal framework.

Dopo questa descrizione ad alto livello, il seguito del capitolo descriverà più in dettaglio le varie fasi del framework.

4.2.2 Feature model

Nella sezione precedente è stato introdotto il feature model e il suo ruolo all'interno del framework per la modellazione della variabilità. In particolare, la caratteristica fondamentale del FM è la sua capacità di variare la propria struttura a livello runtime. La variazione è determinato dallo stato che le feature assumono a runtime (attivo o inattivo), il FM sarà composto solo dall'insieme delle feature attive.

Di conseguenza è cruciale il criterio di determinazione dello stato delle feature, che dovrà essere legato al contesto di esecuzione, affinché la struttura si adatti correttamente al contesto. Per creare la dipendenza tra lo stato delle feature ed il contesto, vengono utilizzate le *precondizioni*. Una precondizione è composta da un nome e un criterio di valutazione. Una precondizione è soddisfatta se la sua valutazione è positiva (restituisce

'true'). Il criterio di valutazione processa le condizioni del contesto di esecuzione. Un esempio di precondizione può essere *hasCamera*, in questo caso la precondizione è soddisfatta se il dispositivo ha una fotocamera funzionante. Le feature possono avere più precondizioni collegate tra loro attraverso operatori logici booleani. Se l'espressione restituisce un valore negativo ('false') lo stato della feature assume il valore "inattivo", e a livello runtime la parte dell'albero corrispondente alla feature ed i suoi figli vengono eliminati.

Il soddisfacimento delle precondizioni è un requisito necessario affinché la feature sia attiva, ma ci sono altri vincoli da soddisfare. Nel seguito della sezione verrà descritto in dettaglio la struttura del FM, insieme alle possibili variazioni a runtime, quindi la modalità di attivazione e disattivazione delle feature (per la rappresentazione del FM, viene utilizzato una notazione ispirato a quella in FODA[32]).

Struttura di una singola feature

Una feature è composta da due parti (Figura 4.2): il suo nome; e l'espressione corrispondente alle sue precondizioni.
Il nome è l'identificativo della feature, serve a distinguere le feature l'una

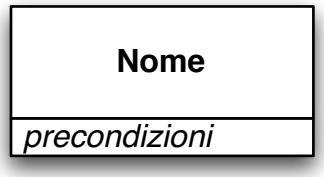

Figura 4.2: Modello di una feature

dall'altra, quindi non possono esserci più feature con lo stesso nome. La

parte delle precondizioni è opzionale, e nel caso in cui non viene specificata, vuol dire che è sempre soddisfatta.

Figli semplici

Le feature possono avere delle feature figlie. Una feature "figlia" può essere *obbligatoria* oppure *opzionale* (Figura 4.3). Nella figura abbiamo che

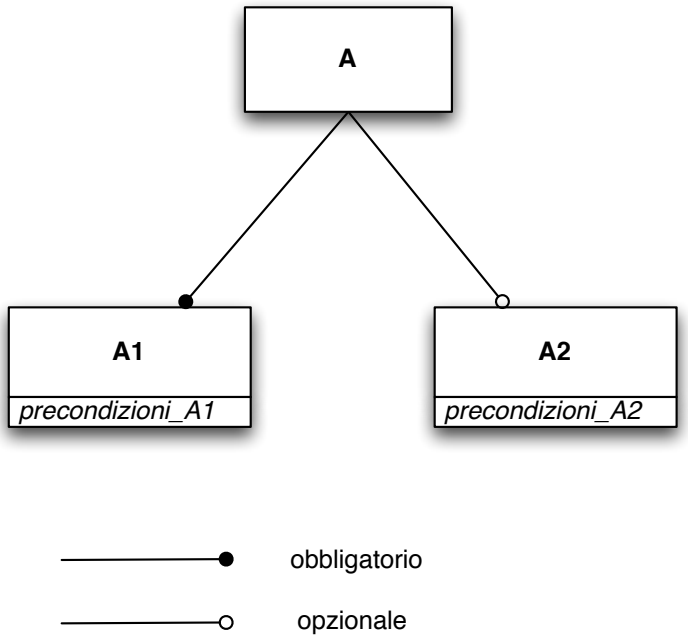

Figura 4.3: Figli di una feature

la feature A1 è obbligatoria, mentre la feature A2 è opzionale. Una feature figlia obbligatoria significa che a livello runtime la feature deve essere sempre attiva affinché l'applicazione funzioni correttamente. Se ha delle precondizioni, i casi in cui queste precondizioni non vengono soddisfatti, il framework solleva un errore. Una feature opzionale vuol dire invece che a runtime la feature potrebbe non essere attiva (sempre dipendente

dalle sue precondizioni), il fatto di essere opzionale fa sì che l'applicazione funzioni lo stesso, semplicemente la funzionalità corrispondente non sarà disponibile.

La feature opzionale è adatta per rappresentare funzionalità dell'applicazione che potrebbero non essere disponibile a seconda delle caratteristiche del dispositivo

Vincoli tra feature

Oltre al legame di padre-figlio, esistono un altro tipo di legame tra feature, chiamato "vincoli tra feature". I vincoli sono di due tipi (figura 4.4): vincoli *requires* e vincoli *excludes*. Il vincolo di *requires* indica che la feature per essere attiva, oltre al soddisfacimento delle sue precondizioni, ha bisogno che anche l'altra feature sia attiva. Mentre il vincolo di *excludes* significa che quando la feature è attiva, l'altra feature non può essere attivata, anche se le sue precondizioni sono soddisfatte. Nella fig. 4.4, A1 per essere attiva ha bisogno che anche A3 sia attiva, mentre se A1 è attiva, A2 non può essere attivata.

Archi *xor*

Oltre ai figli obbligatori oppure opzionali, una feature può anche avere un insieme di figli che sono collegati tra loro attraverso un *arco*.

Un arco *xor* (figura 4.5) indica che tra i figli che appartengono all'arco, solo un figlio può essere attivo. Il criterio di scelta è il seguente:

- tra le feature figlie vengono selezionate le feature le quali sono soddisfatti le precondizioni e i vincoli tra feature;

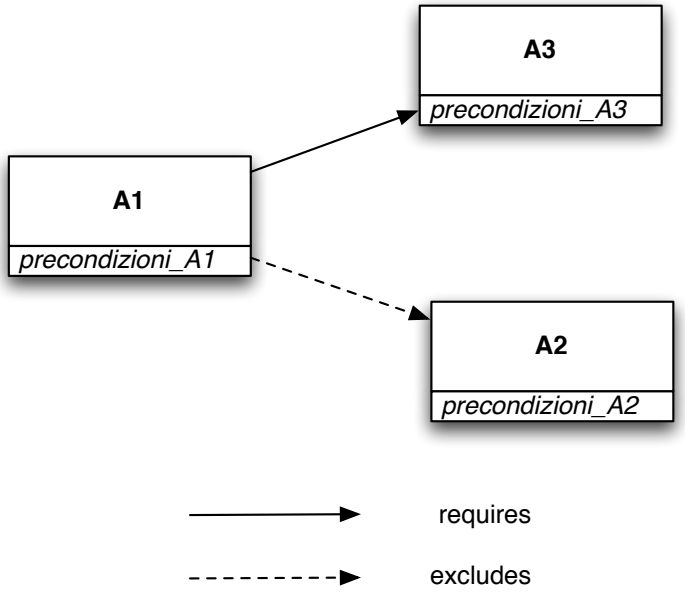

Figura 4.4: Vincoli tra feature

- nel caso il cui ci siano più feature, viene selezionata la feature che appartiene al ramo più a sinistra;
- se invece non c'è alcuna feature con precondizioni soddisfacenti, il framework solleva un'eccezione.

Nell'esempio mostrato nella figura 4.5, verranno verificate le precondizioni di A1 e A2: se entrambe le precondizioni non sono soddisfatte, l'applicazione lancia un errore; nel caso in cui solo una precondizione è soddisfatta, la feature corrispondente verrà attivata; mentre se entrambe le precondizioni sono soddisfatte, viene attivato A1 poiché è la feature che è collocata più a sinistra.

Un arco xor è utile nel rappresentare diverse esecuzioni alternative dell'applicazione, quindi a seconda delle caratteristiche del dispositivo, verrà selezionato l'esecuzione corrispondente.

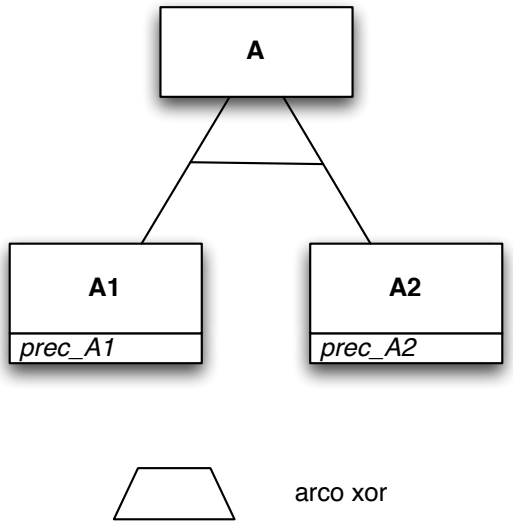

Figura 4.5: Arco xor di un feature model

Archi *or*

Nella figura 4.6 è presente il modello di un arco *or*. Graficamente, un arco è una linea orizzontale che collega i rami corrispondenti all'insieme delle feature figlie, formando un arco colorato. Il significato di un arco or è che tra le feature appartenenti all'arco, possono essere attivate un numero di feature pari alla cardinalità del arco (nella fig. 4.6 possono essere attivate un numero compresso tra 1 e il numero complessivo delle feature appartenenti all'arco). Anche in questo caso, l'attivazione di una feature dipende dalle sue precondizioni. Nel caso il numero delle feature con precondizioni soddisfacenti siano minore della cardinalità minima, l'applicazione lancerà un errore; mentre se il numero delle feature supera la cardinalità massima, vengono scartate le feature collocate più a destra, le quali avranno uno stato inattivo.

Gli archi or è utile per le funzionalità dove potrebbe avere una o più ca-

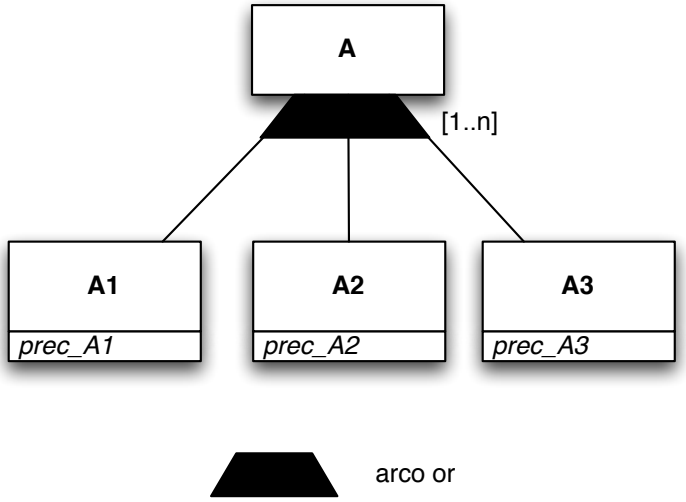

Figura 4.6: Arco or di un feature model

ratteristiche, sempre dipendente dal dispositivo. Un esempio potrebbe essere la funzionalità di condivisione su social network, dove potrebbe essere disponibile la condivisione su Facebook, Twitter, o altri social network, a seconda se è presente la corrispondente applicazione sul dispositivo.

4.2.3 Linguaggio

Il feature model è un elemento fondamentale del framework SAMA. Il feature model però è un modello che si basa su una notazione grafica [32], più comprensibile per fini non-tecnici, però sicuramente più difficile da gestire e utilizzare rispetto ad una notazione testuale, dal momento che la maggior parte degli strumenti di programmazione si basano sui testi. Pertanto, c'è bisogno di un linguaggio che riesca a rappresentare le caratteristiche più importanti del feature model.

Questa sezione descrive la sintassi del linguaggio, e le caratteristiche

del parser che serve a validarlo.

Si è scelto di creare un linguaggio ad-hoc ispirandosi a diversi linguaggi DSML (Domain-Specific Modelling Language) già esistenti [47, 56, 48].

Sintassi

La sintassi del linguaggio è composto da due parti. La prima parte riguarda la definizione della gerarchia delle feature (una rappresentazione dell'albero): vengono specificati i nomi delle feature, le eventuali precondizioni, e le relazioni di padre-figlio. Mentre la seconda parte serve a definire i vincoli tra feature (*requires* ed *excludes*).

Nel corso del capitolo viene utilizzata una grammatica EBNF per una migliore comprensione della sintassi.

Il linguaggio comincia con la dichiarazione del nodo radice(listato 4.1).

```
HIERARCHY : 'application' ROOT_NAME ':' CHILDLIST ';' ;
ROOT_NAME :     //testo
```

Listing 4.1: Grammatica EBNF per la definizione della radice

La dichiarazione comincia con la parola chiave '**application**' seguita dal nome della radice. Dopo i due punti '**:**' comincia la definizione dei figli (**CHILDLIST**) di primo livello (i figli della radice). La definizione termina con il punto e virgola '**;**'

```
CHILDLIST:      (SIMPLE_CHILD | XOR | OR)* ;
```

Listing 4.2: Grammatica EBNF per la definizione dei figli

Ogni figlio può essere:

- figlio semplice, obbligatorio oppure opzionale (**SIMPLE_CHILD**);
- arco xor (**XOR**);
- arco or (**OR**);

Per i *figli semplici* viene specificata solo il nome (**FEATURE_NAME**) se è *obbligatorio*, mentre se è *opzionale* davanti a nome del figlio vi è la parola chiave '**opt**'.

```
SIMPLE_CHILD:   FEATURE_NAME | 'opt' FEATURE_NAME ;
FEATURE_NAME:   //testo
```

Listing 4.3: Grammatica EBNF per i figli semplici

Un *arco xor* comincia con la parola chiave '**one**' seguita dalle parentesi. Dentro le parentesi ci sono i nomi delle feature che fanno parte dell' arco.

```
XOR: 'one' '(' FEATURE_NAME (',' FEATURE_NAME)* ')' ;
FEATURE_NAME:   //testo
```

Listing 4.4: Grammatica EBNF per archi xor

Mentre l'*arco or* ha come parole chiave '**some**'. Dopo la parola chiave c'è la cardinalità dell'arco. La cardinalità è specificata dentro delle parentesi quadre, dove il minimo e il massimo sono separate dai '**..**'.

```
OR:      'some' CARDINALITY '(' FEATURE_NAME (','
           FEATURE_NAME)* ')' ;
CARDINALITY:     '[' NUMBER '..' (NUMBER | 'N' ) ']' ;
FEATURE_NAME:    //testo
NUMBER:          //numero intero
```

Listing 4.5: Grammatica EBNF per archi or

Dopo la definizione della radice, vengono dichiarati gli altri feature che fanno parte del feature model.

```
APPLICATION: 'application' ROOT_NAME ':' CHILDLIST ';' FEATURES;
FEATURES: FEATURE+ ;
```

Listing 4.6: Grammatica EBNF per la definizione della radice

La dichiarazione di una feature comincia con la parola chiave '**feature**', seguito da eventuali precondizioni. Se la feature ha degli figli, allora dopo i due punti '**:**' vi è la definizione della lista dei figli (**CHILDLIST**, come per la radice). La definizione termina con il punto e virgola '**;**'

```
FEATURE: 'feature' FEATURE_NAME PRECONDITIONS? (':' CHILDLIST)?
         ';' ;
```

Listing 4.7: Grammatica EBNF per la definizione di una feature

La definizione dei *vincoli tra feature* comincia con la parola chiave '**constraints**', seguito poi dai vincoli di *requires* (**REQUIRES**) e di *excludes* (**EXCLUDES**)

```
CONSTRAINTS: 'constraints' ':' REQUIRES* EXCLUDES*
```

Listing 4.8: Grammatica EBNF per la definizione dei vincoli

Il vincolo di *requires* è composto dal nome della prima feature, la parola chiave '**requires**', e il nome della seconda feature.

```
REQUIRES : FEATURE_NAME 'requires' FEATURE_NAME ';'
```

Listing 4.9: Grammatica EBNF per i vincoli requires

Il vincolo di *excludes* ha invece come parola chiave '**excludes**'.

```
EXCLUDES : FEATURE_NAME 'excludes' FEATURE_NAME ';'
```

Listing 4.10: Grammatica EBNF per vincoli excludes

Parser: ANTLR

Il linguaggio che è stato creato serve al programmatore per modellare il feature model della sua applicazione. Una volta che il programmatore ha definito il suo feature model, quindi scritto il linguaggio corrispondente, il framework lo utilizzerà per le fasi successive. Però come tutti gli input, prima di essere analizzato, c'è bisogno di un controllo sulla correttezza sintattica. Il controllo verrà effettuato da un *parser*.

Anche se il parser è trasparente al programmatore, il ruolo del parser è molto importante nel framework. Oltre a controllare la sintassi del

linguaggio, il parser costruisce anche l'*Abstract Syntax Tree* (AST) del linguaggio, che verrà utilizzato successivamente dal framework per generare i componenti della logica applicativa.

Il parser del framework è stato costruito utilizzando *ANTLR* [57]. ANTLR (ANother Tool for Language Recognition) è uno strumento generatore di parser e traduttori che permette di definire grammatiche sia nella sintassi ANTLR (simile alla EBNF e a quella di YACC) sia in una speciale sintassi astratta per *AST* (Abstract Syntax Tree). ANTLR può creare scanner, parser e AST. ANTLR è tuttavia qualcosa di più di un linguaggio di definizione di grammatiche, gli strumenti forniti permettono di implementare la grammatica definita in ANTLR generando automaticamente scanner, e parser (o tree parser) sia in Java, C++ oppure Sather.

ANTLR implementa una strategia di analisi sintattica PRED-LL(k) e consente un lookahead di lunghezza arbitraria per disambiguare nei casi di ambiguità.

Ricordiamo velocemente come funziona un parser. Il parser (o analizzatore sintattico) ha bisogno di un *lexer* (analizzatore lessicale).
Il compito del lexer è impacchettare il flusso di caratteri (altrimenti) insignificante in gruppi che, elaborati dal parser, acquistano significato. Ogni carattere o gruppo di caratteri raccolto in questo modo viene detto token. Il lexer converte il flusso di caratteri in un flusso di token che hanno un significato individuale stabilito dalle regole del lexer. Il flusso di token generato dal lexer viene ricevuto in ingresso dal parser. Un lexer di solito genera errori riguardanti le sequenze dei caratteri che non riesce a mettere in corrispondenza con gli specifici tipi di token definiti dalle sue regole.
I linguaggi sono descritti attraverso grammatiche. Una grammatica determina esattamente quello che definisce un dato token e quali sequenze di

token sono considerate valide. Il parser organizza i token che riceve in sequenze ammesse definite della grammatica del linguaggio, e controlla che si conformi alla sintassi del linguaggio definito dalla grammatica. Se il linguaggio viene usato esattamente come definito nella grammatica, il parser sarà in grado di riconoscere i pattern che costituiscono specifiche strutture del discorso e raggrupparle insieme opportunamente. Se il parser incontra una sequenza di token che corrisponde a nessuna delle sequenze consentite, solleverà un errore e forse proverà a recuperarlo facendo alcune assunzioni sulla natura di tale errore.

La grammatica per la costruzione del parser del nostro linguaggio è mostrato nel listato 4.11.

```
application
        :       'application' ID ':' childList ';' features? constraints? ;
childList
        :       child (',' child)* ;
child
        :       one
        |       some
        |       featureList ;
one
        :       'one' '(' featureList ')' ;
some
        :       'some' cardinality '(' featureList ')' ;
featureList
        :       (ID | optFeature )  (',' ID | optFeature)* ;
optFeature
        :       'opt' ID  ;
features
        :       feature+ ;
feature
        :       'feature' ID ('(' preconditions? ')')? (':' childList)? ';'  ;
preconditions
        :       condition  ;
condition
        :       (('('condition')')|('~'? ID)) (('^'|'v') condition)? ;
```

```
cardinality
        :       '[' min '..'  max']'  ;
min     :       INT  ;
max     :       INT
        |       '*'  ;
constraints
        :       'constraints' ':' constraint+  ;
constraint
        :       ID 'requires' ID ';'
        |       ID 'excludes' ID ';'  ;


ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
INT :   '0'..'9'+ ;
WS  :   ( ' ' | '\t' | '\r' | '\n' ) {$channel=HIDDEN;} ;
```

Listing 4.11: Grammatica ANTLR per la costruzione del parser

Come si può notare, la grammatica ANTLR è molto simile ad una grammatica EBNF, quindi per quanto riguarda la sintassi della grammatica, valgono tutte le regole delle grammatiche EBNF. Nella grammatica le regole con il nome in minuscolo corrispondono alle regole sintattiche (regole per il parser), mentre le regole in maiuscolo sono le regole lessicali (regole per il lexer).

Una volta definito la grammatica, ANTLR genera automaticamente una classe Java per il lexer, e una classe Java per il parser. Per convalidare il linguaggio del feature model, il framework invoca il metodo *validate* della classe Parser. Il frammento del codice che viene utilizzato è mostrato nel listato 4.12, supponendo che il linguaggio del feature model sia contenuto in un file esterno.

```
try{
        Lexer lexer = new Lexer(new ANTLRFileStream(languageFileName));
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        Parser parser = new Parser(tokens);
        parser.validate(languageFileName);
```

```
}catch(Exception e){
        e.printStackTrace();
}
```

Listing 4.12: Frammento di codice per convalidare il linguaggio

Il parser, oltre ad effettuare il controllo sintattico, effettua anche i seguenti controlli:

- non esistano feature con lo stesso nome, quindi feature duplicati;
- per ogni feature che viene dichiarata, verifica che la feature fa parte dell'albero, cioè deve essere figlia di un'altra feature oppure della radice.

La grammatica oltre a creare il parser (e anche il corrispondente lexer), permette anche di creare il *Abstract Syntax Tree*(AST). L' Abstract Syntax Tree è un elemento molto importante per il framework, poiché organizza il linguaggio analizzato in una struttura ad albero, cioè ricrea una rappresentazione del feature model a partire dal linguaggio. Questo sarà fondamentale per la fase successiva, quella della generazione delle classi (sezione 4.2.4).

La costruzione del AST viene fatto nella grammatica, per la cronaca nel listato 4.13 viene illustrato la grammatica completa anche delle regole di costruzione del AST. La spiegazione delle regole di costruzione del AST non verrà effettuata in questa tesi, maggiori informazioni si può trovare sul sito di ANTLR [57].

```
application
        :       'application' ID ':' childList ';' features? constraints? ->
            ^(ID childList features? constraints?) ;
childList
        :       child (',' child)* -> ^(CHILDREN child+);
```

```
child   :       one
        |       some
        |       featureList ;
one     :       'one' '(' featureList ')' -> ^(ONE ^(CHILDREN featureList))
        ;
some
        :       'some' cardinality '(' featureList ')' -> ^(SOME cardinality
            ^(CHILDREN featureList)) ;
featureList
        :       (ID | optFeature )  (','! ID | optFeature)* ;
optFeature
        :       'opt' ID -> ^(ID OPT) ;
features:       feature+ -> ^(FEATURES feature+)
        ;
feature
        :       'feature'! ID^  ('(' preconditions? ')')? (':'! childList)? ';'!
             ;
preconditions
        :       condition  -> ^(PRECONDITIONS condition) ;
condition
        :       (('('condition')')|('~'? ID)) (('^'|'v') condition)? ;
cardinality
        :       '['!min '..'! max']'! ;
min     :       INT -> ^(MIN INT) ;
max     :       INT -> ^(MAX INT)
        |       '*' -> ^(MAX '*') ;
constraints
        :       'constraints' ':' constraint+ -> ^(CONSTRAINTS constraint+) ;
constraint
        :       ID 'requires' ID ';' -> ^(REQUIRE ID+)
        |       ID 'excludes' ID ';' -> ^(EXCLUDE ID+) ;


ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
INT :   '0'..'9'+ ;
WS  :   ( ' ' | '\t' | '\r' | '\n' ) {$channel=HIDDEN;} ;
```

Listing 4.13: Grammatica ANTLR per la costruzione del parser con le regole di AST

4.2.4 Meccanismo di generazione

Una volta creato il feature model, il framework basa la struttura della logica applicativa sul FM, sfruttando così la sua caratteristica di riuscire ad adattarsi al contesto di esecuzione. In particole, la struttura della logica applicativa viene generata a partire proprio dal feature model. L'elemento che si occupa della generazione delle classi è il *generator*. La generazione delle classi parte dal linguaggio corrispondente al feature model. Più precisamente, il generator prende come input il "Abstract Syntax Tree" (AST) creato dal parser (sezione 4.2.3).

L'AST contiene informazione riguardo la gerarchia delle feature, e i vincoli tra feature. Il processo di generazione consiste in una scansione dell'AST, partendo dalla radice fino a visitare intero albero.

Il framework è stato sviluppato per applicazioni mobili appartenenti alla piattaforma Android, che utilizza il linguaggio di programmazione Java. I componenti generati corrisponderanno a classi Java. Questa sezione descrive in dettaglio questo meccanismo di generazione, con le classi Java generate.

Annotazioni

Innanzitutto vengono generate delle classi annotazioni che verranno successivamente utilizzate da altre classi Java generate.

Una classe annotazione generata è *Precondition* (listato 4.14).

L'annotazione Precondition è un'annotazione di classe, il suo ruolo è quello di memorizzare le informazioni riguardanti le precondizioni di una feature, e i vincoli di 'requires' ed 'excludes'.

Un'altra classe annotazione che viene generata è la classe *Cardinality* (listato 4.16).L'annotazione Cardinality è un'annotazione che viene utiliz-

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Precondition{
        String conditions();
        String requires();
        String excludedBy();
}
```

Listing 4.14: Annotazione Precondition

zato sulle variabili, serve a definire i valori minimi e massimi di un arco 'or'(quando il valore massimo è 'N', in Cardinality il campo max assume il valore di '-1').

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Cardinality {
        public int min();
        public int max();
}
```

Listing 4.15: Annotazione Cardinality

Poi vi è l'annotazione *Nullable*, utilizzato sulle variabili. Serve a tener traccia delle feature figlie opzionali. Questo significa che il campo su cui è annotato a livello runtime può assumere il valore *null*.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Nullable {
}
```

Listing 4.16: Annotazione Nullable

Archi xor ed or

Durante la visita dell'albero, ogni volta che si incontra un arco xor oppure or, viene generata una classe astratta. Il nome della classe astratta è composto dal nome della feature padre dell'arco concatenato con un suffisso. Il suffisso per un arco xor è '**_one**', mentre il suffisso per un arco or è '**_some**'. Nel listato 4.17 ci sono esempi classi astratte degli archi xor ed or.

```
public abstract class ParentFeatureName_one{
}

public abstract class ParentFeatureName_some{
}
```

Listing 4.17: Classi astratte corrispondenti agli archi xor ed or

Features

Per ogni feature viene generata una classe Java (figura 4.18). Il nome della classe è uguale al nome della feature. La classe è annotata con l'annotazione *Precondition* generata in precedenza (sezione 4.2.4), che contiene le informazioni riguardo le precondizioni della feature e i vincoli 'requires' ed 'excludes'. Se la feature fa parte di un arco xor oppure or, allora la classe Java estende la classe astratta corrispondente all'arco. Per le variabili d'istanza della classe, ci sarà una corrispondenza diretta con la lista dei figli della feature:

- ogni feature figlia semplice corrisponde ad una variabile d'istanza del tipo della classe corrispondente alla feature figlia. Se la feature figlia è opzionale, la variabile corrispondente sarà annotata dalla classe annotazione *Nullable*;

- per un arco xor viene generata una variabile d'istanza del tipo della classe astratta corrispondente all'arco (sezione 4.2.4)
- per un arco or viene generata una collezione del tipo della classe astratta corrispondente all'arco.

```
@Precondition(conditions="conditions",
    requires="featuresRequired",excludedBy="featuresExcludedBy"}
public class MyFeatureName extends ParentFeatureName_one{
        private SimpleChildName simpleChildName;

        @Nullable
        private OptionalSimpleChildName optionalSimpleChildName;

        private MyFeatureName_one myFeatureName_one;

        @Cardinality(min=1, max=-1)
        private Set<MyFeatureName_some> listMyFeatureName_some;

        @Inject
        public MyFeatureName(....){
                this.simpleChildName = simpleChildName;
                this.optionalSimpleChildName = optionalSimpleChildName;
                this.myFeatureName_one = myFeatureName_one;
                this.listMyFeatureName_some = listMyFeatureName_some;
        }

        ...
}
```

Listing 4.18: Esempio di classe Java generata a partire da una feature

Da notare che il costruttore è annotato con *@Inject*. L'annotazione Inject è un'annotazione delle librerie GUICE [58], e serve per il Dependency Injection a livello runtime (sezione 4.2.5).

ConditionVerifier e Module

Una classe molto importante che viene generata è *ConditionVerifier*. Questa classe ha il compito di determinare lo stato delle feature (quindi se la classe Java corrispondente può essere istanziata). Il metodo da utilizzare è *isValid(Precondition precondition)*, che prende come parametro l'oggetto Precondizione della classe Java corrispondente. Più precisamente, il metodo valuta l'espressione corrispondente alle precondizioni della classe, poi effettua i controlli sui vincoli tra feature.
Per quanto riguarda la valutazione delle precondizioni, ConditionVerifier utilizza un database costituito da coppie di nome e criterio di valutazione della precondizione. Quindi, per ogni precondizione, ConditionVerifier esegue il criterio corrispondente. Se il nome di una precondizione non rientra nel database, la precondizione non è mai soddisfatta. Il framework offre già una lista di precondizioni con il nome e criterio di valutazione (tabella 4.1). Il programmatore può aggiungere altre precondizioni alla lista, inserendo il nome e il pezzo di codice per la valutazione.

Module è una classe generata che viene utilizzata dall'oggetto *injector* della libreria *GUICE* [58]. Injector è l'oggetto responsabile dell'istanziazione degli oggetti Java secondo il pattern Dependency Injection (vedi sezione 4.2.5). Nella classe Module vengono generate le regole per l'Injection. Questi saranno strettamente corrispondenti alle regole di attivazione delle feature. Injector utilizza la classe ConditionVerifier per decidere se istanziare l'oggetto o meno (sezione 4.2.5).

Nome	Criterio di valutazione
hasCamera	il dispositivo ha una fotocamera funzionante
hasAutoFocus	la fotocamera del dispositivo è auto focus
hasInternet	il dispositivo è connesso in rete
hasBluetooth	il dispositivo ha la funzionalità di bluetooth attivato
hasGPS	il dispositivo ha GPS attivato
hasMicrophone	il dispositivo ha a disposizione un microfono
hasTouchScreen	lo schermo del dispositivo è di tipo Touch Screen
hasUSB	il dispositivo supporta connessione USB
lowBattery	il livello di batteria del dispositivo è basso

Tabella 4.1: Lista delle precondizioni attualmente supportate dal framework

4.2.5 Runtime: Injection

Fin'ora si è discusso dell'approccio a livello di progettazione, partendo dalla definizione del feature model, fino ad arrivare alla generazione della struttura logica dell'applicazione. Si è detto che l'obiettivo del framework è quello di dare alle applicazioni mobili un comportamento adattivo, quindi in questa sezione verrà decritto il comportamento di un'applicazione a livello runtime, focalizzando su come il framework gestisce gli adattamenti al contesto.

La gestione dell'adattamento avviene attraverso una manipolazione della struttura della logica applicativa. La logica applicativa deriva dal feature model, quindi ha una struttura ad albero. In particolare, allo startup di un'applicazione, viene istanziata la classe Java corrispondente alla feature radice. La classe Java istanzierà le sue variabili (riferimenti alle feature figlie), quindi con la radice, vengono istanziate anche le classi corri-

spondenti alle sue feature figlie. Le feature figlie a loro volta istanzieranno le loro figlie, quindi dalla radice viene creata tutta la struttura della logica applicativa.

L'adattamento consiste nella modalità di creazione degli oggetti Java. La logica applicativa deve essere strettamente legata al feature model, per sfruttare la sua caratteristica di adattarsi al contesto. Quindi così come il feature model è composto solo dalle feature attive, a livello runtime vengono istanziati solo oggetti delle classi Java corrispondenti alle feature attive. Per ottenere questa corrispondenza tra oggetti Java istanziati e feature attive, la creazione degli oggetti deve essere basato su delle regole. Non si usa più con il classico operatore Java "new", ma si utilizza la *Dependency Injection*.

Dependency Injection è un *design pattern*. L'idea di base è quella di avere un componente esterno (injector) che si occupi della creazione degli oggetti e delle loro relative dipendenze e di assemblarle mediante l'utilizzo dell'injection basate su specifiche regole. Il framework utilizza le librerie GUICE per l'implementazione della Dependency Injection. Un esempio di frammento di codice per l'istanziazione è mostrato nel listato 4.19.

```
try{
        Injector injector = Guice.createInjector(new Module());
        RootFeature root = injector.getInstance(RootFeature.class);
}catch(Exception e){
        e.printStackTrace();
}
```

Listing 4.19: Frammento di codice per istanziare oggetti attraverso le librerie GUICE

Nelle librerie GUICE, l'elemento che si occupa di istanziare oggetti Java è l'oggetto *Injector*. Le regole di injection sono contenute nella classe

Module, che viene generata dal framework insieme alle classi Java corrispondenti alle feature, durante la scansione del Abstract Syntax Tree.

Le regole di injection si basano sui criteri di determinazione dello stato di una feature. La classe Java viene istanziata se la feature corrispondente è attiva. Nel feature model la determinazione dello stato di una feature consiste nella verifica delle sue precondizioni e i vincoli tra feature. Queste informazioni sono contenute nell'annotazione *Precondition*, mentre la verifica viene eseguita dall'oggetto *ConditionVerifier*, generato dal framework (sez. 4.2.4). La verifica avviene con il metodo *isValid(Precondition precondition)*, e consiste innanzitutto nella valutazione delle precondizioni, se sono soddisfatti viene effettuato anche la verifica dei vincoli. Per quanto riguarda le precondizioni, per ogni nome c'è un criterio di valutazione. Il framework genera, nella classe ConditionVerifier i criteri di valutazione corrispondenti alle precondizioni della tabella 4.1, il programmatore a sua volta potrà aggiungere o modificare nuove precondizioni con il rispettivo criterio di verifica.

A partire dalla radice, ogni classe Java corrispondente ad una feature ha il compito di istanziare le proprie figlie. In particolare, l'injector "inietta" l'oggetto "giusto" alla variabile corrispondente. Le regole di injection delle variabili consistono in:

- variabili corrispondenti a figli semplici: se la feature corrispondente è attiva, alla variabile viene assegnato un' istanza della classe corrispondente , mentre se non è attiva viene assegnato *null*. Da notare, se la feature figlia è obbligatoria, quindi se non c'è l'annotazione @Nullable, assegnando il valore null, il framework lancerà un'eccezione. Il programmatore poi potrà gestire l'eccezione a suo modo;
- variabili corrispondenti ad archi xor (ad un arco xor corrisponde una

classe astratta): tra le feature appartenenti ad un arco xor, solo una feature può essere attiva, quindi alla variabile verrà assegnato un' istanza della sottoclasse corrispondente alla feature attiva;

- variabili corrispondenti ad archi or (ad un arco or corrisponde una classe astratta): la variabile corrispondete è una collezione di oggetti della classe astratta corrispondente, tra tutte le sottoclassi viene aggiunto un' istanza alla collezione se la feature corrispondete è attiva.

In questo modo, si è riuscito a creare un corrispondenza univoca tra la struttura del feature model e la struttura della logica applicativa, quindi così come il FM dipende dal contesto, la stessa cosa vale per la logica applicativa, dando così all'applicazione questo comportamento adattivo.

4.2.6 Conclusioni

Il comportamento adattivo di un'applicazione è determinato dal fatto che gli oggetti, che servono a manipolare l'esecuzione dell'applicazione, non sono sempre gli stessi, ma cambia a seconda del contesto di esecuzione. Il framework si basa su questo concetto, utilizzando il feature model per modellare questa dipendenza tra oggetti e il contesto. Per esempio, una variabile corrispondente ad un arco xor (di tipo di una classe astratta), a seconda del soddisfacimento delle precondizioni, può contenere un oggetto di una sottoclasse C1 oppure di una sottoclasse C2, con C1 e C2 che hanno due esecuzioni diverse. Siccome la valutazione delle precondizioni dipende dalle caratteristiche del dispositivo, siamo riusciti ad adattare il comportamento dell'applicazione secondo le caratteristiche del dispositivo.

Nel capitolo 5 verrà esemplificato tutte le fasi di progettazione del framework, e il comportamento a runtime, attraverso un caso di studio.

4.3 Vantaggi

Questa sezione descrive i principali vantaggi nell'utilizzo del framework per il processo di sviluppo di un'applicazione mobile.

Come è stato ripetuto più volte nella tesi, il principale vantaggio dell'approccio è quello di riuscire ad adattare l'applicazione al tipo di dispositivo in cui è installato. L'adattamento del framework SAMA deriva dalla caratteristica del feature model di poter variare la propria struttura a livello runtime. Per esempio, quando ci sono diverse alternative di esecuzione, questo può essere rappresentato da un arco xor. La selezione della feature attiva (quindi la corrispondente alternativa di esecuzione) verrà eseguito dal framework a livello runtime. Quindi la gestione consiste solo nel definire le feature appartenenti all'arco, e se in futuro si vuole aggiungere una nuova alternativa, basta aggiungere una nuova feature all'arco con le sue precondizioni.

Un altro vantaggio è quello di *disaccoppiare la progettazione con l'implementazione*. Con il framework SAMA si riesce ad avere una separazione tra la fase di progettazione dell'applicazione e la sua implementazione: la fase di progettazione consiste nel definire il feature model dell'applicazione; una volta definito il feature model, il programmatore implementa le classi Java generate dal framework. Nella fase di progettazione, lo sviluppatore può per esempio concentrarsi solo sulla lista di funzionalità che l'applicazione deve avere, ignorando come verranno implementati (attraverso componenti ad-hoc, invocando servizi esterni, o utilizzando applicazioni

di terze parti, etc.). La scelta sarà effettuata dopo nel momento in cui si andrà ad implementare le classi Java generate.

Con l'utilizzo di oggetti feature per rappresentare le funzionalità dell'applicazione, e lasciando al framework la decisione riguardo l'istanziazione della classe Java corrispondente, si riesce ad evitare un codice contorto con cascate di if-else per la gestione delle esecuzioni alternative. Quindi con il framework SAMA il codice è più leggibile e da più facile da mantenere.

Inoltre, rappresentando le varie funzionalità con oggetti feature, SAMA migliora la *riusabilità* del codice, dal momento che la stessa funzionalità (quindi la rispettiva feature) può essere facilmente riutilizzato da altre applicazioni. Questo vantaggio è fondamentale per accorciare il ciclo di vita dello sviluppo, che è cruciale nel mondo mobile.

Capitolo 5

Caso di studio: Winery

Nel capitolo 4 è stato descritto il framework SAMA. Questo capitolo illustrerà un esempio di utilizzo del framework per un'applicazione mobile. L'applicazione in questione è *Winery*, un'applicazione android.

5.1 Descrizione

Winery è un'applicazione semplice riguardante i vini, offre una funzionalità di ricerca a partire dal nome di un vino. Più precisamente, l'utente inserisce il nome, l'applicazione effettua una ricerca online a partire dal nome, e restituisce una lista di vini con quel nome. Una volta ottenuto la lista dei vini, l'utente può selezionare un vino per avere le informazioni riguardanti il vino (colore, prezzo medio, storia, etc.). L'applicazione inoltre offre anche una funzionalità di condivisione delle informazioni sui social network (Facebook, Twitter, etc.). Oltre la funzionalità di ricerca a partire dal nome, è possibile effettuare una ricerca a partire anche da un codice a barre. In questo caso l'applicazione utilizza la fotocamera del dispositivo: l'utente inserisce l'immagine del codice a barre; a partire dall'imma-

gine l'applicazione prima ricava il numero del codice a barre corrispondente; poi dal numero ricava il nome del prodotto corrispondente per poi effettuare la ricerca online.

Nel seguito di questo capitolo mostreremo le fasi di progettazione dell'applicazione Winery con il framework SMA ed il suo comportamento a runtime.

5.2 Progettazione

Il framework prevede le seguenti fasi per la progettazione dell'applicazione mobile: (1) progettazione della feature model che modella il comportamento adattivo dell'applicazione e (2) l'implementazione delle classi Java generate dal framework a partire dal feature model.

5.2.1 Architettura generale

L'architettura dell'applicazione è stato costruito secondo il pattern MVC (Figura 5.1).

Nel *Model* ci sono le classi per la gestione dei dati. In Winery Model è composto da un'unica classe *Wine* per modellare un oggetto di tipo Vino. Mentre nella *View* ci sono le *Activity* dell'applicazione, i quali nel hanno il ruolo di essere delle viste, quindi serve come interfaccia grafica con l'utente. Nel *Controller* è contenuto la logica applicativa dell'applicazione generata dal feature model.

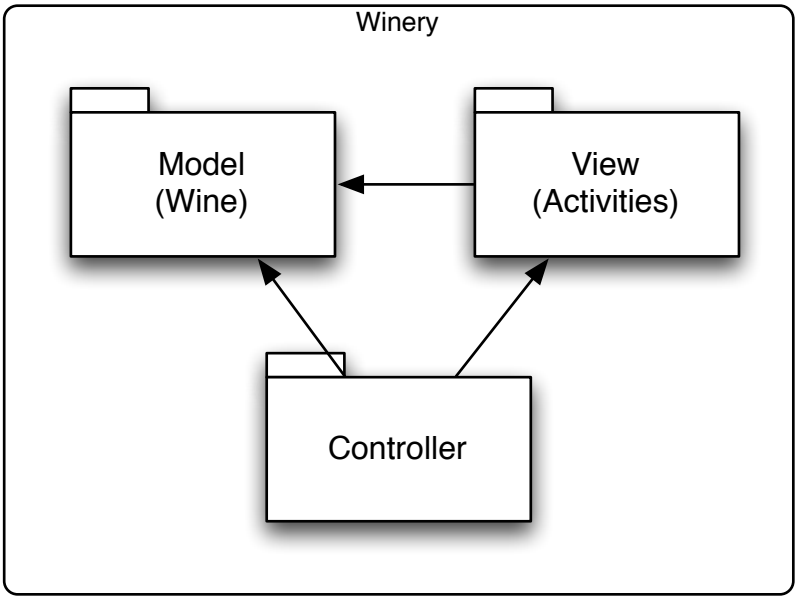

Figura 5.1: Architettura dell'applicazione Winery

5.2.2 Feature model e il linguaggio corrispondente

Il framework prevede innanzitutto la progettazione del feature model. La struttura del feature model di Winery è rappresentato nella figura 5.2, mentre nella tabella 5.1 ci sono le descrizioni delle feature. L'applicazione è composta da tre principali funzionalità: *GetName*, *WebSearch*, *Share*. GetName è la funzionalità che permette di ricavare il nome del vino su cui effettuare la ricerca. Una volta ottenuto il nome, l'applicazione utilizza la funzionalità *WebSearch* per effettuare la ricerca online. WebSearch, per poter essere disponibile, ha bisogno che il dispositivo abbia attiva una connessione internet (*hasInternet*). Una volta ottenuto le informazioni riguardante un vino, l'applicazione offre la funzionalità *Share*, che permette all'utente di condividere le informazioni sui social network.

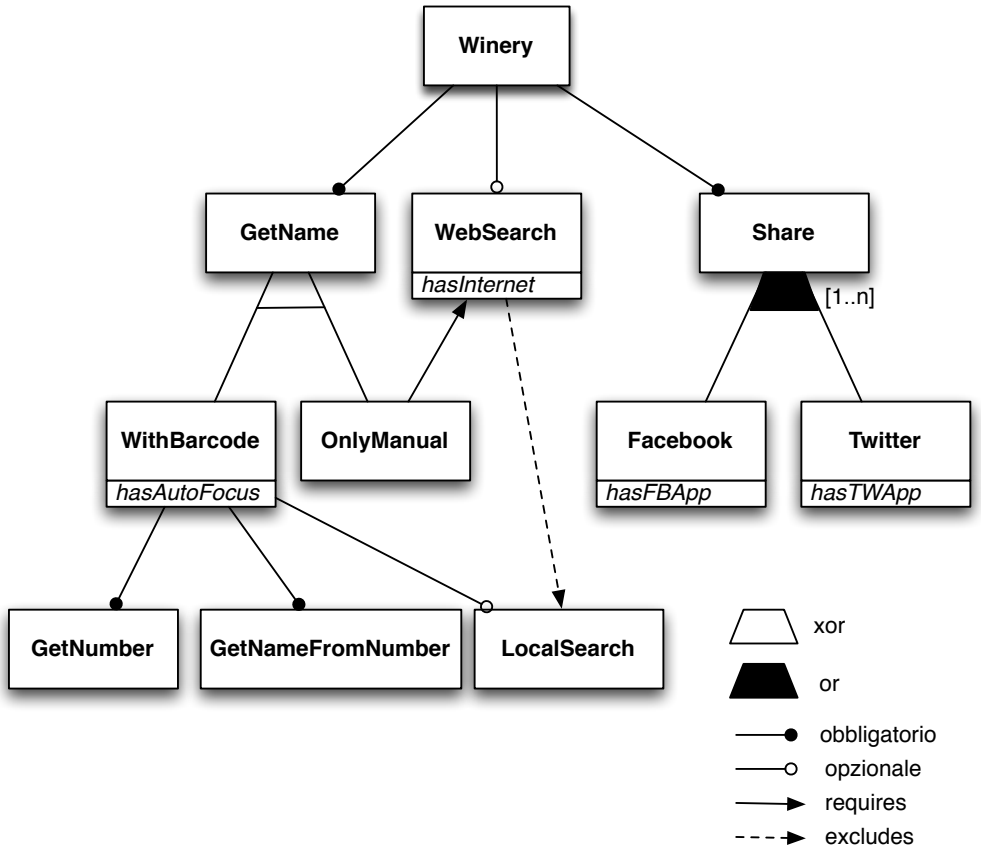

Figura 5.2: Feature model di Winery

Nome	**Descrizione**
Winery	l'applicazione Winery
GetName	acquisisce il nome da ricercare
WithBarcode	c'e' la funzionalità di ricerca a partire dal codice a barre, oltre alla ricerca dal nome
OnlyManual	c'è solo la funzionalità di ricerca dal nome
GetNumber	recupera il numero dall'immagine del codice a barre
LocalSearch	ricerca del vino in locale
WebSearch	ricerca online dei vini a partire dal nome
Share	permette all'utente di condividere le informazioni riguardante un vino sui social network
Facebook	condivisione delle informazioni di un vino sul social network Facebook
Twitter	condivisione delle informazioni di un vino sul social network Twitter

Tabella 5.1: Le feature dell'applicazione Winery

Il nome su cui effettuare la ricerca può essere ottenuto da input di tastiera oppure da una immagine di un codice a barre. In particolare, se la fotocamera del dispositivo è di tipo auto focus (*hasAutoFocus*), nell'applicazione saranno presenti entrambi le opzioni, mentre se non è di tipo auto focus, sarà presente solo l'input da tastiera. Nel feature model questo si traduce in un arco xor sotto la funzionalità GetName. L'arco xor è composto da *WithBarcode*, che rappresenta entrambe le opzioni, e *OnlyManual* che indica che è presente solo l'input da tastiera. Dal momento che WithBarcode ha la precedenza, se le sue precondizioni sono soddisfatte, allora sarà attivo, altrimenti si attiva OnlyManual(la parte precondizioni è vuota, quindi è sempre soddisfatta). Quindi se il dispositivo ha la fotocamera auto focus(*hasAutoFocus*) allora sono presente entrambe le opzioni (WithBarcode), altrimenti c'è solo input da tastiera(OnlyManual).
La funzionalità di ricerca da codice a barre prevede di tradurre l'immagine del codice a barre al corrispondente numero (*GetNumber*), poi ricavare il nome del prodotto dal numero di codice a barre (*GetNameFromNumber*). Nel caso in cui il dispositivo non è connesso in rete, l'applicazione prevede una ricerca nella memoria locale (*LocalSearch*) che contiene i risultati delle recenti ricerche effettuate. Questo nel feature model corrisponde al vincolo di *excludes* da WebSearch a LocalSearch.
Il vincolo *requires* da OnlyManual a WebSearch è determinato dal fatto che non essendo disponibile la ricerca locale, c'è bisogno obbligatoriamente di una ricerca online.
La funzionalità Share consiste nella condivisione delle informazioni sui social network. In particolare, si potrà condividere su Facebook se sul dispositivo è presente un'applicazione Facebook (*hasFBApp*), Twitter se il dispositivo ha un'applicazione Twitter (*hasTWApp*). Nel feature model que-

sto corrisponde ad un arco or con cardinalità da 1 a n, e composto dalle feature *Facebook* e *Twitter*, con le rispettive precondizioni.

Una volta progettato il feature model, il passo successivo è di rappresentarlo con il linguaggio del framework. Il linguaggio corrispondente al feature model di Winery è rappresentato nel listato 5.1(per la sintassi del linguaggio, si veda la sezione 4.2.3).

```
application Winery : GetName, opt WebSearch, Share;
feature GetName: one(WithBarcode, OnlyManual);
feature WithBarcode(hasAutoFocus): GetNumber, GetNameFromNumber, opt LocalSearch;
feature GetNumber;
feature GetNameFromNumber;
feature LocalSearch;
feature OnlyManual;
feature WebSearch(hasInternet) ;
feature Share : some[1..n](Facebook, Twitter);
feature Facebook (hasFBApp);
feature Twitter(hasTWApp);

constraints:
OnlyManual requires WebSearch;
WebSearch excludes LocalSearch;
```

Listing 5.1: Linguaggio corrispondente al feature model di Winery

5.2.3 Classi Java generate

Una volta modellato il feature model, quindi tradotto nel rispettivo linguaggio, il framework lo utilizza per generare le classi Java corrispondenti. Le classi Java generate saranno poi implementate dal programmatore secondo le sue specifiche sull'esecuzione dell'applicazione. Il diagramma delle classi generate è mostrato nella figura 5.3.

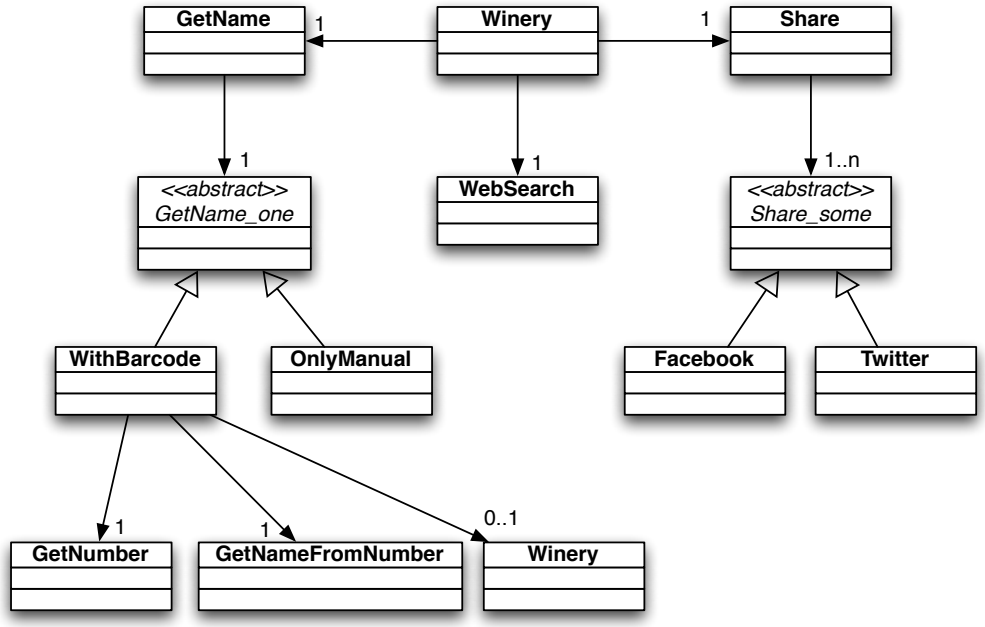

Figura 5.3: Diagramma delle classi della logica applicativa di Winery

Per ogni feature del feature model, il framework genera una classe Java corrispondente. Quindi abbiamo la classe *Winery* che corrisponde alla radice. Winery è l'unica classe visibile al di fuori del Controller, quindi fa da interfaccia alle altre parti dell'architettura. Nella classe Winery sono contenute tutte le operazioni necessarie per l'esecuzione dell'applicazione.
Per lo svolgimento delle operazioni, ogni feature utilizza i suoi figli, quindi Winery ha riferimento alle classi *GetName*, *WebSearch*, *Share*, e li invocherà per svolgere le sue operazioni. GetName ha un arco xor. Per gli archi xor e/o or, il framework genera una classe astratta corrispondente con il nome del padre più il suffisso "_one" per xor, "_some" per or. Quindi GetName ha riferimento ad un oggetto della classe astratta *GetName_one*.
Le classi corrispondenti a feature appartenenti ad un arco estendono la classe astratta corrispondente all'arco, quindi *WithBarcode* e *OnlyManual*

estendono GetName_one. Le precondizioni di WithBarcode saranno memorizzate nell'annotazione di classe *Precondition*, cosi come il vincolo "requires" di OnlyManual. WIthBarcode poi avrà riferimenti ai suoi figli, tra cui *LocalSearch* che sarà annotato con *Nullable* poiché corrisponde ad un figlio opzionale.
Per quanto riguarda *Share*, avrà una collezione di oggetti della classe astratta *Share_some*. Infine le classi *Facebook* e *Twitter* estendono Share_some.

Nel listato 5.2 c'è un esempio di classe generata (nel esempio c'è la classe WithBarcode).

```
@Precondition(conditions="hasAutoFocus",requires="",excludedBy="")
public class WithBarcode extends GetName_one{

        private GetNumber getNumber;

        private GetNameFromNumber getNameFromNumber;

        @Nullable
        private LocalSearch localSearch;

        @Inject
        public Barcode(GetNumber getNumber,GetNameFromNumber getNameFromNumber,
            LocalSearch localSearch){
                this.getNumber = getNumber;
                this.getNameFromNumber = getNameFromNumber;
                this.localSearch = localSearch;
        }

        ......

}
```

Listing 5.2: Classe Java WithBarcode

Infine, ricordiamo che viene generato anche la classe *Module* che contiene le regole per la Dependency Injection a livello runtime.

5.3 Runtime

Nella sezione 5.2 è stato descritto le fasi di progettazione di Winery utilizzando il framework SAMA. In questa sezione analizzeremo il comportamento dell'applicazione a livello runtime.

Allo startup di un'applicazione Android, viene lanciata la prima classe Activity chiamata *MainActivity*. La MainActivity invocherà la classe Java Winery chiedendogli di settare i parametri (in questo caso, l'interfaccia grafica). Il primo elemento che serve all'applicazione è il nome su cui effettuare la ricerca, quindi Winery chiede a GetName di settare la funzionalità per ottenere il nome. GetName a sua volta chiamerà l'oggetto GetName_one.
In questo passaggio si può osservare una caratteristica molto importante del framework. Nel framework SAMA, la creazione degli oggetti corrispondenti alle feature viene fatta attraverso la Dependency Injection basandosi su delle regole di iniezione. In SAMA le regole vengono generate a partire dal feature model basandosi sulle regole di attivazione delle feature. Nel nostro caso, all'oggetto di tipo GetName_one (che è una classe astratta), verrà iniettata un'istanza di WithBarcode oppure un'istanza di OnlyManual, a seconda delle regole generate. Più precisamente, viene fatta una verifica sulla fotocamera del dispositivo, se è di tipo auto focus, allora viene creata un istanza di WithBarcode, altrimenti viene creata un'istanza di OnlyManual. L'istanza di WithBarcode creerà sia la funzione di ricerca da codice a barre sia l'input da tastiera (utilizzando gli oggetti corrispondenti ai suoi figli), mentre l'istanza di OnlyManual creerà solo la funzione di input da tastiera.
Quindi questo è un esempio di dipendenza tra l'esecuzione dell'applicazione e le caratteristiche del dispositivo, facendo si che l'applicazione si

adatti al dispositivo in cui è installato (infatti, a secondo del tipo di fotocamera del dispositivo, ci sarà o meno la funzionalità di ricerca da codice a barre). La cosa fondamentale è che tutto questo viene fatto dal framework a livello runtime, senza che il programmatore debba implementarlo a livello statico, magari con l'uso di rami if-else, rendendo il codice poco leggibile e mantenibile.

Continuando con l'esecuzione dell'applicazione, una volta ottenuto il nome su cui effettuare la ricerca, Winery lo passa come parametro all'oggetto WebSearch, il quale effettuerà la ricerca online.
Nel mostrare le informazioni riguardante il vino corrispondente, Winery evocherà l'oggetto Share di settare le funzionalità di condivisione. Share a sua volta utilizzerà la sua lista di oggetti Share_some. La lista verrà istanziata sempre attraverso la Dependency Injection (con le regole di injection). In particolare, se nel dispositivo è installato un'applicazione facebook, la lista conterrà un oggetto della classe Facebook, se è installato un'applicazione twitter, ci sarà un oggetto della classe Twitter. Quindi se il dispositivo ha un'applicazione facebook, allora si potrà condividere le informazioni sul social network Facebook evocando l'applicazione corrispondente, mentre se è installato un'applicazione twitter, allora sarà possibile condividere su Twitter, sempre invocando l'applicazione corrispondente.

Capitolo 6

Conclusioni e lavoro futuro

In questo lavoro di tesi è stato presentato il framework SAMA, un framework per lo sviluppo di applicazioni mobili. L'obiettivo del framework è di riuscire a dare all'applicazione un comportamento variabile che si adatta al contesto di esecuzione. A differenza dei tradizionali approcci, SAMA riesce a separare la logica applicativa dell'applicazione con la logica di adattamento, evitando così di introdurre rami if-else nella logica applicativa, con risultato di un codice più leggibile e più facile da mantenere.

Il framework si basa sui principi del Dynamic Software Product Lines, e consiste nel organizzare l'architettura dell'applicazione basandosi su una struttura ad albero, chiamato feature model. Il feature model poi viene rappresentato da un linguaggio DSML, e utilizzato dal framework per generare i componenti dell'applicazione. Mentre a livello runtime il comportamento variabile è gestito attraverso l'utilizzo della Dependency Injection.

L'approccio è stato testato attraverso lo sviluppo di un'applicazione Android le cui funzionalità dipendono dalle caratteristiche del dispositivo

su cui è installata.

Possibili sviluppi consiste in: testare il framework su applicazioni per tablet; supportare anche applicazioni mobili non appartenenti alla piattaforma Android (e.g., applicazioni iOS); costruzione di un IDE, possibilmente integrato con strumenti comunemente utilizzati come Eclipse, per facilitare la progettazione del feature model (magari con una traduzione automatica dal feature model al linguaggio corrispondente).

Bibliografia

[1] Betty HC Cheng, Rogerio De Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap*. Springer, 2009.

[2] Philip K McKinley, Seyed Masoud Sadjadi, Eric P Kasten, and Betty HC Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[3] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[4] Linda Northrop and P Clements. Software product lines, 2001.

[5] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

[6] Rick Rogers, John Lombardo, Zigurd Mednieks, and Blake Meike. *Android application development: Programming with the Google SDK*. O'Reilly Media, Inc., 2009.

[7] twicca. http://twicca.r246.jp.

[8] Josh Dehlinger and Jeremy Dixon. Mobile application software engineering: Challenges and research directions. In *Workshop on Mobile Software Engineering*, 2011.

[9] Tony Wasserman. Software engineering issues for mobile application development. *FoSER 2010*, 2010.

[10] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c. In *Software Language Engineering*, pages 246–265. Springer, 2011.

[11] Christine Julien and G-C Roman. Egospaces: Facilitating rapid development of context-aware mobile applications. *Software Engineering, IEEE Transactions on*, 32(5):281–298, 2006.

[12] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *Software Engineering, IEEE Transactions on*, 29(10):929–945, 2003.

[13] Malte Appeltauer, Robert Hirschfeld, and Tobias Rho. Dedicated programming support for context-aware ubiquitous applications. In *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM'08. The Second International Conference on*, pages 38–43. IEEE, 2008.

[14] Bart Van Wissen, Nicholas Palmer, Roelof Kemp, Thilo Kielmann, and Henri Bal. Contextdroid: an expression-based context framework for android. *Proceedings of PhoneSense*, 2010.

[15] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. Modeling and validating dynamic adaptation. In *Models in Software Engineering*, pages 97–108. Springer, 2009.

[16] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *Software, IEEE*, 23(2):62–70, 2006.

[17] Julian Ohrt and Volker Turau. Cross-platform development tools for smartphone applications. 2012.

[18] Andrew Lunny. *PhoneGap Beginner's Guide*. Packt Publishing, 2011.

[19] Boydlee Pollentine. *Appcelerator Titanium Smartphone App Development Cookbook*. Packt Publishing Ltd, 2011.

[20] Dipanjan Chakraborty, Anupam Joshi, Tim Finin, and Yelena Yesha. Service composition for mobile environments. *Mobile Networks and Applications*, 10(4):435–451, 2005.

[21] Gianpaolo Cugola, Carlo Ghezzi, Leandro Sales Pinto, and Giordano Tamburrelli. Adaptive service-oriented mobile applications: A declarative approach. In *Service-Oriented Computing*, pages 607–614. Springer, 2012.

[22] Nelly Bencomo, Pete Sawyer, Gordon Blair, and Paul Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *2nd International Workshop on Dynamic Software Product Lines (DSPL 2008), Limerick, Ireland*, volume 38, page 40, 2008.

[23] Gilles Perrouin, Franck Chauvel, Julien DeAntoni, Jean-Marc Jézéquel, et al. Modeling the variability space of self-adaptive applications. In *2nd Dynamic Software Product Lines Workshop (SPLC 2008, Volume 2)*, pages 15–22, 2008.

[24] Svein Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using product line techniques to build adaptive systems. In *Software Product Line Conference, 2006 10th International*, pages 10–pp. IEEE, 2006.

[25] Reinhard Wolfinger, Stephan Reiter, Deepak Dhungana, Paul Grunbacher, and Herbert Prahofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*, pages 21–30. IEEE, 2008.

[26] Jaejoon Lee and Kyo Chul Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Software Product Line Conference, 2006 10th International*, pages 10–pp. IEEE, 2006.

[27] Kurt Geihs, Mohammad Ullah Khan, Roland Reichle, Arnor Solberg, Svein Hallsteinsen, and Simon Merral. Modeling of component-based adaptive distributed applications. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 718–722. ACM, 2006.

[28] Ildefonso Montero, Joaquín Peña, and Antonio Ruiz-Cortés. Representing runtime variability in business-driven development systems. In *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*, pages 228–231. IEEE, 2008.

[29] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. A dynamic software product line approach using aspect models at runtime. In *5th Domain-Specific Aspect Languages Workshop*, 2010.

[30] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, et al. K@ rt: An aspect-oriented and model-oriented framework for dynamic software product lines. In *Proceedings of the 3rd International Workshop on Models@ Runtime, at MoDELS'08*, 2008.

[31] Kwanwoo Lee, Kyo C Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Software Reuse: Methods, Techniques, and Tools*, pages 62–77. Springer, 2002.

[32] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[33] Mark Simos, R Creps, Carol Klingler, and L Lavine. Software technology for adaptable reliable systems (stars). organization domain modeling (odm) guidebook, version 1.0. Technical report, DTIC Document, 1995.

[34] Martin L Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the rseb. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 76–85. IEEE, 1998.

[35] Martin L Griss. Software reuse architecture, process, and organization for business success. In *Computer Systems and Software Engineering, 1997., Proceedings of the Eighth Israeli Conference on*, pages 86–89. IEEE, 1997.

[36] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-; oriented reuse method

with domain-; specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

[37] Kwanwoo Lee, Kyo C Kang, Wonsuk Chae, and Byoung Wook Choi. Feature-based approach to object-oriented engineering of applications for reuse. *Software-Practice and Experience*, 30(9):1025–1046, 2000.

[38] Kwanwoo Lee, Kyo C Kang, Eunman Koh, Wonsuk Chae, Bokyoung Kim, and Byoung Wook Choi. Domain-oriented engineering of elevator control software. In *Software Product Lines*, pages 3–22. Springer, 2000.

[39] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, Sabine Moisan, Jean-Paul Rigault, et al. Modeling context and dynamic adaptations with feature models. In *Proceedings of the 4th International Workshop Models@ run. time*, 2009.

[40] Carlos Cetina, Joan Fons, and Vicente Pelechano. Applying software product lines to build autonomic pervasive systems. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 117–126. IEEE, 2008.

[41] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Using feature models for developing self-configuring smart homes. In *Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on*, pages 179–188. IEEE, 2009.

[42] Pablo Trinidad, Antonio Ruiz-Cortés, Joaquın Pena, and David Benavides. Mapping feature models onto component models to build

dynamic software product lines. In *International Workshop on Dynamic Software Product Line, DSPL*, 2007.

[43] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. Fama: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 129–134, 2007.

[44] Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72. ACM, 2004.

[45] Danilo Beuche. Modeling and building software product lines with pure:: variants. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 358–358. IEEE, 2008.

[46] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing tvl, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS?10), Linz, Austria, January*, pages 27–29, 2010.

[47] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143, 2011.

[48] Arie Van Deursen and Paul Klint. *Domain-specific language design requires feature descriptions*. CWI, 2001.

[49] Don Batory. *Feature models, grammars, and propositional formulas*. Springer, 2005.

[50] Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762. ACM, 2009.

[51] Marcılio Mendonça. *Efficient reasoning techniques for large scale feature models*. PhD thesis, University of Waterloo, 2009.

[52] Guus Schreiber, Bob Wielinga, Hans Akkermans, Walter Van de Velde, and Anjo Anjewierden. Cml: The commonkads conceptual modelling language. In *A future for Knowledge Acquisition*, pages 1–25. Springer, 1994.

[53] Andreas Abele, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, David Servat, Martin Törngren, and Matthias Weber. The cvm framework: A prototype tool for compositional variability management. In *Proceeding of: Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, pages 101–105, 2010.

[54] Steven She and Thorsten Berger. Formal semantics of the kconfig language. *Technical note, University of Waterloo*, page 24, 2010.

[55] Android. http://www.android.com.

[56] Kacper Bak. Clafer: a unified language for class and feature modeling. Technical report, Technical report, Generative Software Development Lab, 2010.

[57] Antlr. http://www.antlr3.org.

[58] Guice. http://code.google.com/p/google-guice/.