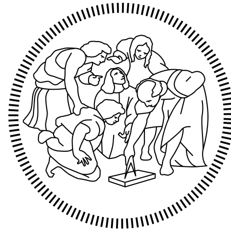
POLITECNICO DI MILANO

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica



Polo Territoriale di Como

Model Driven Data Synchronization for Mobile Applications

Relatore: Prof. Piero FRATERNALI
Correlatore: Prof. Marco BRAMBILLA

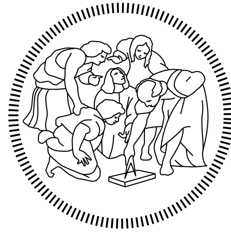
Tesi di laurea di:
Jacopo MOSSINA
Matr. 804790

Anno Accademico 2014 - 2015

POLITECNICO DI MILANO

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica



Polo Territoriale di Como

Model Driven Data Synchronization for Mobile Applications

Relatore: Prof. Piero FRATERNALI
Correlatore: Prof. Marco BRAMBILLA

Tesi di laurea di:
Jacopo MOSSINA
Matr. 804790

Anno Accademico 2014 - 2015

Acknowledgments

Sentiti ringraziamenti vanno al Prof. Piero Fraternali, Relatore, al Prof. Marco Brambilla, Correlatore e all'Ing. Felix Javier Acero Salazar, Assistente: senza il loro assiduo supporto e la loro guida sapiente, questo lavoro non sarebbe stato realizzabile.

Un ringraziamento particolare va poi a WebRatio s.r.l, per l'interesse dimostrato verso questa attività di ricerca e per il supporto concreto, manifestato anche attraverso la concessione di software in fase di sviluppo.

Alla mia famiglia
Ai miei amici
A Francesca

Abstract

The pervasiveness of mobile applications in the software engineering industry has been introducing a remarkable set of problems, among which the deployment-related ones stand out, due to the multitude of platforms and devices to serve. The Model Driven Engineering (MDE) approach, proposing a platform independent design methodology taking on these difficulties, requires enhanced models and sophisticated code generation techniques. In this model-centered study, we aim at introducing new design patterns joining the concepts of User Experience (UX) and software business logic: data synchronization patterns associated to interaction. The transverse nature of these artifacts allows us to feature them in front-end designs models, augmenting their expressive power and providing continuity with the back-end logic. As the definition of patterns requires a profound preliminary examination, our research starts by analyzing some complex real-world applications to infer the triggering dynamics of data synchronization, underscoring their implications on user interaction. Eventually, we present some interaction-centered patterns derived from the results of this inquiry. Subsequently, the main thread of our work deviates to data synchronization in mobile applications, introducing its core aspects, different specializations and their adoption criteria. Since the domain-specific complexities have already inspired the conception of data synchronization patterns for mobile applications in literature, our objective is to combine these patterns with the outcomes of the aforementioned examination over user interaction. The resulting artifacts of this stage are finally shown at work into a real-world application modeling scenario, followed by a detailed explanation of design choices, impact on expressive power of the model and encountered or potential issues.

Sommario

La pervasività delle applicazioni mobili nell'industria dell'ingegneria del software ha causato l'introduzione di un notevole insieme di problematiche, tra cui spiccano quelle relative al deployment, a causa della moltitudine di piattaforme e dispositivi da servire. L'approccio Model Driver Engineering, fondato su una metodologia di design indipendente dalla piattaforma in grado di affrontare tali criticità, richiede modelli avanzati e sofisticate tecniche di generazione di codice. In questo studio, incentrato sulla modellazione, miriamo ad introdurre nuovi pattern di design capaci di conciliare concetti di User Experience (UX) e logica di business, ovvero pattern di sincronizzazione dati associati all'interazione. La natura trasversale di questi artefatti ci permette di integrarli nei modelli di design front-end, potenziandone la forza espressiva e introducendo continuità con la logica di back-end. Dal momento che la definizione dei pattern esige un'approfondita esaminazione preliminare, la nostra ricerca è avviata dall'analisi di alcune applicazioni complesse già presenti sul mercato, atta alla deduzione delle dinamiche scatenanti degli eventi di sincronizzazione dati, con particolare attenzione alle implicazioni sull'interazione dell'utente. Di seguito, presentiamo alcuni pattern centrati sull'interazione, ottenuti dalla valutazione dei risultati dell'indagine. Successivamente, il filo conduttore del lavoro devia verso la sincronizzazione dati in applicazioni mobili, introducendone gli aspetti fondamentali e le diverse specializzazioni associate ai rispettivi criteri di adozione. Poiché le complessità specifiche di tale dominio hanno già ispirato la concezione di pattern di sincronizzazione dati per applicazioni mobili, la nostra ambizione consiste nel combinare questi pattern con quelli ottenuti dalla già citata esaminazione sull'interazione dell'utente. Gli artefatti risultanti da questo stadio sono infine mostrati nell'integrazione in uno scenario reale di modellazione software, accompagnato da spiegazioni esaustive su scelte di progettazione, impatto sulla potenza espressiva del modello e problematiche incontrate o potenziali.

Contents

Introduction **1**

1 Analysis of Data Sync in Commercial Applications **5**
1.1 Motivation and Goals . . . 5
1.2 Modus Operandi . . . 6
1.3 Methodology . . . 6
1.4 Case Study 1: Evernote . . . 7
1.4.1 Triggering Dimension . . . 7
1.4.2 Storage and Transfer Dimensions . . . 7
1.4.3 Special Features . . . 7
1.4.4 Behavioral analysis . . . 8
1.5 Case study 2: Dropbox . . . 10
1.5.1 Triggering Dimension . . . 10
1.5.2 Storage and Transfer Dimensions . . . 10
1.5.3 Special Features . . . 11
1.5.4 Behavioral analysis . . . 11
1.6 Case study 3: Instagram . . . 13
1.6.1 Triggering Dimension . . . 13
1.6.2 Storage and Transfer Dimensions . . . 13
1.6.3 Special Features . . . 13
1.6.4 Behavioral analysis . . . 13
1.7 Analysis Report – Tabular View . . . 15

2 User Interaction Patterns **17**
2.1 The Interactional Perspective . . . 17
2.2 Interaction Flow Modeling Language . . . 18
2.3 Patterns Identification and Analysis . . . 19
2.3.1 Pattern: Content Scrolling . . . 20
2.3.2 Pattern: Context Change . . . 21
2.3.3 Pattern: Pull-To-Refresh . . . 23
2.3.4 Pattern: Form Submission . . . 24

2.3.5 Pattern: Application Launch 26

3 Data Synchronization Logic **29**
3.1 Synchronization Types and Elements 30
3.1.1 The Synchronization Grid 30
3.1.2 Change Tracking 32
3.1.3 Conflict Resolution 32
3.2 Client-Server Communication 34
3.2.1 Directional Aspects 34
3.2.2 Temporal Aspects 35

4 Data Synchronization Patterns **37**
4.1 Patterns By Time 40
4.1.1 Asynchronous Data Synchronization 40
4.1.2 Synchronous Data Synchronization 43
4.2 Patterns By Storage Strategy 46
4.2.1 Partial Storage 46
4.2.2 Complete Storage 48
4.3 Patterns By Transfer Logic Sophistication 51
4.3.1 Full Transfer 51
4.3.2 Timestamp Transfer 53
4.3.3 Mathematical Transfer 55

5 Patterns Composition **59**
5.1 Building the Big Picture 59
5.1.1 Tabular Synthesis 60
5.2 Composite Patterns Conception 61
5.2.1 Application Launch Synchronous Sync 62
5.2.2 Content Scrolling Asynchronous Sync 68
5.2.3 Context Change Asynchronous Sync 71
5.2.4 Pull-To-Refresh Asynchronous Sync 74
5.2.5 Form Submission Synchronous Sync 77
5.2.6 Push-Triggered Sync 80

6 Application of the Patterns in Mobile Front-end Modeling **85**
6.1 Software Requirements 85
6.2 Model Realization 87
6.2.1 Design of Data Synchronization Logic 88
6.2.2 Patterns Application 89
6.2.3 Resulting Complete Model 93

7 Conclusions and Future Work **97**

8 Related Work **99**
8.1 Design Patterns in Software Development 99
8.2 MDE, IFML and Mobile Applications 101

Bibliography **105**

List of Figures

2.1 Content Scrolling pattern rendition in IFML 20
2.2 Context Change pattern rendition in IFML 22
2.3 Pull-To-Refresh pattern rendition in IFML 23
2.4 Form submission pattern rendition in IFML 25
2.5 Application launch pattern rendition in IFML 27

4.1 UML s.d. for Asynchronous Data Sync 41
4.2 UML s.d. for Synchronous Data Sync 44
4.3 UML s.d. for Partial Storage 47
4.4 UML s.d. for Complete Storage 49
4.5 UML s.d. for Full Transfer 52
4.6 UML s.d. for Timestamp Transfer 54
4.7 UML s.d. for Mathematical Transfer 56

5.1 IFML d. of Application Launch Synchronous Sync 63
5.2 UML s.d. for Application Launch Synchronous Sync 64
5.3 IFML d. of Application Launch Asynchronous Sync 66
5.4 UML s.d. for Application Launch Asynchronous Sync 67
5.5 IFML d. of Content Scrolling Asynchronous Sync 69
5.6 UML s.d. for Content Scrolling Asynchronous Sync 70
5.7 IFML d. of Context Change Asynchronous Sync 72
5.8 UML s.d. for Context Change Asynchronous Sync 73
5.9 IFML d. of Pull-To-Refresh Asynchronous Sync 75
5.10 UML s.d. for Pull-To-Refresh Asynchronous Sync 76
5.11 IFML d. of Form Submission Synchronous Sync 78
5.12 UML s.d. for Form Submission Synchronous Sync 79
5.13 IFML d. for Push-Triggered Sync 81
5.14 UML s.d. for Push-Triggered Sync 82

6.1 Application Front-end model in IFML 87
6.2 Push-Triggered Sync variant application 89
6.3 Push-Triggered Sync variant application 90

6.4 Application Launch Asynchronous Sync 91
6.5 Application Launch Asynchronous Sync variant application 91
6.6 Combination of C.S. and P.T.R. asynchronous sync 92
6.7 Form Submission Synchronous Sync application 93
6.8 App Front-end model integrating patterns - part 1 94
6.9 App Front-end model integrating patterns - part 2 95

List of Tables

1.1 Triggering Dimension 15
1.2 Storage and Data Transfer Dimensions 15
1.3 Special Features 15

3.1 Grid classifying the existing synchronization technologies 30

5.1 Compatibility between U.I. and D.S. patterns 60

Introduction

The mobile ecosystem is undergoing an outstanding expansion, affecting not only existing industry segments, but planting the seeds for new realities. Enabled by the overcoming of constraints related to computational power, reduced memory and storage of mobile devices, the enterprises exploring the mobile world are challenging the idea of ameliorating users' lives exploiting the possibilities provided by the pervasiveness of mobile devices. New disciplines and trends are rapidly growing on top of this idea, intersecting multiple areas of interests: many leading hardware and software companies are extending, reviewing their research scope to deepen their knowledge and provide better integrated solutions for users in mobility. This enthusiastic impulse has been introducing a huge amount of novelties reshaping the existing ecosystem: let us mention, for instance, the spread of wearable devices and the vision of the Internet of Everything.

On the other hand, the general demand over software quality and complexity has been increased as well. Today, applications designed to run on mobile devices feature rich interfaces implying convoluted interaction mechanisms and impressive business logic implementations, allowed by the technological evolution of the target devices. Additionally, thinking to all of the latest trends characterizing the mobile software environment, like Connected Living, Advertising, Digital Commerce and Security, the relevance of data-intensive services is undeniable.
Critical tasks like data synchronization, contextual information retrieval and the activation of communication paths to access external service providers are difficult to orchestrate, and having to consider them at design time is not a trivial software engineering problem.

In this problematic context, coping with functional and non-functional requirements of mobile applications serving multiple platforms on different devices results in the emergence of the need of a systematic approach. Matter of fact, in the typical front-end development of mobile applications manual coding still remains a predominant practice, with all the liabilities it involves: low reuse, inefficient maintainability, problems of portability, risks of inconsistencies are just some of them.
Approaches like Model Driven Engineering (MDE), conversely, leverage the introduction of the abstraction relying on their core discipline, software modeling. However, using a model driven approach to produce quality software introduces controversies related to the sophistication level of models and the complexity of their implementation through code generation techniques (exploited to produce actual code).

In this thesis we are focusing on the front-end modeling perspective, trying to introduce some novel artifacts to cope with the requirements over data. In particular, data synchronization is a recurring assignment, abstracting a huge amount of scenarios that must be covered by applications relying on remote data access and processing. The approach we adopt is purposely pattern-based, given the correlation between the problem-solving nature of data synchronization and the problem-solution structure of design patterns. Since patterns proposal requires an accurate understanding of the dynamics to represent, we move towards data synchronization with a bottom-up analysis of the state of the art, resulting from the study of real-world applications. The solution that our patterns are meant to provide combines aspects of user interaction with the business logic implied by data synchronization. Given the heterogeneous nature of these artifacts, they are expressed using two different languages: Interaction Flow Modeling Language (IFML), representing front-end concepts, and Unified Modeling Language (UML), modeling the behavior of the application components. Both IFML and UML have been adopted as standards by the Object Management Group (OMG). The most valuable goal we are pursuing with this experimental work is to demonstrate the effectiveness of our introductions in terms of expressive power of front-end models, which are usually not covering transverse processes like data alignment.

The thesis is structured as follows:

- Chapter 1 describes the analysis performed on real-world applications, aiming at the individuation of some standard approaches to data synchronization.
- Chapter 2 focuses on the triggering nature of interaction over data synchronization events, proposing a list of patterns in IFML.
- Chapter 3 illustrates the core elements of data synchronization, emphasizing its communication-related features.
- Chapter 4 presents some known-in-literature patterns on data synchronization, reviewed to adopt a more recognizable notation.
- Chapter 5 illustrates the patterns resulting from the composition of artifacts presented in Chapters 2 and 4.
- Chapter 6 demonstrates the applicability of patterns presented in Chapter 5 in a real-world modeling scenario.
- Chapter 7 comprises the conclusions and the potential future work.

Chapter 1

Analysis of Data Synchronization in Commercial Applications

1.1 Motivation and Goals

In this first chapter of our research study, we are adopting a learning strategy based on observation and analysis. This approach has several advantages: above any other, the reports are not contaminated by literature, previous researches and studies; conversely, they are the authentic outcomes of an experimental work. The need for this kind of analysis as an opening for the subject is witnessed by the lack of documented knowledge on standardized practices for data synchronization in mobile environment, but also by the mutation-prone shaped nature of the environment itself. Furthermore, investigating applications behaviors according to scenarios based on our interest satisfies our need for flexibility over research parameters and method.

In terms of goals, the analysis aims at profiting from the collected observations and aid the definition of some innovative patterns, under two different perspectives: user interaction and synchronization methodology. Additionally, we try to identify some best practices and understand the impact of requirements on both subjects.

1.2 Modus Operandi

The analysis is performed by observing the functionality of some Android commercial applications, whose selection criterion is the likeliness of deducing synchronization mechanisms and their interactional implications. The investigation starts by the most trivial scenarios, like the launch of the application, to evolve towards the study of some significant use cases. To better understand some mechanism involving data transfer, we monitor write operations on the device's internal memory, also relying on "superuser" privileges for a deeper exploration. In addition, we try to infer some unexposed logic by examining the applications' logs at runtime using the Android Device Monitor, a tool included in the Android SDK.

1.3 Methodology

Reporting is organized by separating observations according to their context:

Triggering dimension
Everything that concurs in firing the synchronization event.

Storage and transfer dimensions
How data is replicated, cached and transferred.

Special features
Unique characteristics of the application associated to data synchronization.

Analysis of the application behavior in meaningful use cases
Description of the applications' feedbacks in terms of user experience and "under-the-hood" logic to notable scenarios in which data synchronization is involved.

Finally, the outcomes of the analysis are synthesized into tabular views, to emphasize similarities, identify common practices and ad-hoc features.

1.4 Case Study 1: Evernote

1.4.1 Triggering Dimension

Synchronization happens as a response to some interactions: launching or resuming the application, opening a note, starting or leaving edit mode, deleting a note, adding tags or renaming a notebook and finally, forcing it by touching the dedicated voice in the menu.

Automated synchronization is performed periodically, having the time interval set by the user, who is able to furtherly restrict the process actuation based on the type of network connectivity.

1.4.2 Storage and Transfer Dimensions

Storage strategy. Evernote[1] relies on its own cache portion (where "cache" stands for actual application data), as most Android applications do: data is cached in a directory in the /data partition. Thumbnails and attachments are cached in dedicated subdirectories.

Data transfer. Transfers are ruled by events and interaction, as underscored describing the triggering dimension: at app launch all notes previews are downloaded and displayed in the main view, while actual contents are downloaded (and eventually stored) when opening notes for the first time. Clearly, this design choice aims at providing the best user experience possible, trying to minimize latency times.

1.4.3 Special Features

Offline mode. The above-mentioned storage strategy is sensible, because it allows working with stale data without connectivity, which is almost crucial in a note-taking framework. Intuitively, storing the entire dataset would be even more convenient under this perspective, but it would be too much demanding in terms of storage.

Clean/Dirty flags. By inspecting the log, we note that changes are reflected on all nodes using "clean/dirty" strategy, presumably recalling the snooping protocols mechanics used in cached-based multiprocessors. This suggests that, probably,

[1] Published by Evernote Corporation, version 6.3.3.1, tested on Android 4.4.4

when a resource is modified within a node it is flagged as dirty, so that any other linked endpoint notifies the change when synchronizing data.

1.4.4 Behavioral analysis

Scenario 1: Data Read at Mobile-side
By Logcat inspection, we observe that "EvernoteProvider" (which is, intuitively, an Android Content Provider instance) object is designed to perform CRUD[2] and control operations on files. Resources are accessed by default in read mode: to enter the edit mode, there is a dedicated button triggering the UI and logic change. From this simple use case, let us observe that each resource accessed is read from the local storage: it seems to be a design choice to empower offline mode and to grant UI responsiveness.

Scenario 2: Posting (Data Created or Modified at Mobile-side)
The editor works in offline mode, allocating a draft copy in cache, without triggering events until the user chooses to save the note (and, subsequently, leaves the editor returning to the list of notes). When this happens, the draft is saved locally and, if network connectivity is available, a synchronization process starts: most likely, the operations are performed in asynchronous way, with visual feedbacks in the UI. In particular, notes whose modifications are to be updated are flagged with a dedicated icon, while the running synchronization process is emphasized by an animated indeterminate progress-bar under the application ActionBar[3]. Before synchronizing, the application makes sure it already has the latest version of the edited note: this operation prevents conflictual updates and seems to be asymmetric with respect to the logic implemented by Evernote web application (refer to scenario 4 for conflictual editing). The sync process works as follows:

1. Upload metadata (notebooks, tags changes)
2. Upload actual note changes
3. Request lock to perform transaction
4. Open (or create) remote note file
5. Copy draft note content to remote note file and to local cache (to ensure consistency)
6. Mark note as dirty on server (to allow update propagation on other linked devices)

[2]Acronym for Create Read Update Delete.
[3]Name for the standard application menu-bar in Android.

7. Release lock and refresh the UI

Scenario 3: Data Created or Modified via Backend Services
Modifying a note via another client, being it the web one or a desktop variation, does not affect the behavior of the mobile application at all. Let us underscore that no push notifications seem to be adopted to guarantee real-time updates: rather, the synchronization triggering mechanism provided is a combination of interaction-based and automated. The latter handles remote changes as well, capturing conflicts and determining the policy to solve them. Summarizing, assuming that network connectivity is available, resources are updated re-launching the application or forcing the synchronization manually, otherwise not intervening forces to wait for the automated process to start.

Scenario 4: Data Conflicts Induction
As a note is accessed in edit mode, the application generates a new empty draft and stores it within its dedicated cache. Then the draft is filled with the existing content of the note, which the "EvernoteProvider" reads from the local copy (i.e. from the device storage): this is very interesting, because a concurrent modification of the same resource from another client results in a conflict, managed by note duplication. This scenario is easily verifiable: both resources are successfully saved, but the most recently submitted one gets the prefix "Conflict note" before its actual name. The saving process is thoroughly described in scenario 2. The story is different when deleting an entry via mobile application: the resource is obviously deleted on the server, so any concurrent changes to the same resource coming from other clients are discarded. If the deleted resource was already outdated, instead, the delete operation erases only the outdated note (on the local cache), but the subsequent synchronization event catches the new version and performs a simple transfer – in practice, deletion of outdated entries prevents merges and conflicts.

1.5 Case study 2: Dropbox

1.5.1 Triggering Dimension

Synchronization is triggered by both navigation events and actions performed on data, locally. Dropbox[4] for Android is essentially a client structured as a fully functional file manager, thus exploration is based on a layered (tree-based) structure, in which every change of context (i.e. browsing directory) requires the knowledge of the internal structure of the context itself (content and subdirectories). Data refresh is performed for each navigation event. Local actions on data are triggers for synchronization processes, since they are handled as transactions: CRUD operations fire data refresh. Additionally, on-demand data synchronization is available as an option in the UI, and data refresh is performed at each application launch.

Automated Synchronization via “long polling” (inferred via log inspection): it is a variation of the traditional polling technique, still categorized as pull technology, since the initial request for data originates from the client, and then is responded to by the server. Long polling allows emulating a push mechanism under circumstances where such push is not possible, such as sites with security policies that require rejection of incoming HTTP/S requests. Functionally, the client requests information from the server exactly as in traditional polling, except it issues its requests (polls) at a much slower frequency. If the server does not have any information available for the client when the poll is received, instead of sending an empty response, it holds the request open and waits for response information to become available: once it does, the server immediately sends a network response to the client, completing the open request. Long polling not only has the advantage of consistently reducing the frequency of requests over polling, but it permits the elimination of the response latency associated to such technique, due to the time interval between requests.

1.5.2 Storage and Transfer Dimensions

Storage strategy. At the very first launch the application performs the setup of its caching structure, which relies on both private data (path: /data/com.dropbox.android/app_DropboxSyncCache/<id>/) and internal storage (path: /storage/Android/data/com.dropbox.android/cache/<id>/) resources: the former stores databases, while the latter is intended to store actual files.

[4]Published by Dropbox Inc., version 2.4.9.00, tested on Android 5.0.2

Data transfer. It essentially exploits timestamps, but it is ruled by precise policies that appear to be differentiated based not only on user interaction, but also on data type. Pre-fetching is performed to ensure an acceptable tradeoff between data availability and user experience, following the logic of look-ahead applied on user interaction.

1.5.3 Special Features

Offline mode. The offline experience is very limited: read operations are allowed given that target data is already cached, data creation fails (even though it appears to be designed to resume when connectivity is available) while deletion is denied.

1.5.4 Behavioral analysis

Scenario 1: Data Read at Mobile-side
SSL sockets are opened to establish the connection required by long polling technique. The log also exposes the start of a service named “DbxSyncService”, as well as the functional logic of long polling: while Dropbox is running (in foreground or background), a request is periodically sent to the server (approximately one every 30 seconds). When connectivity drops, to each long polling failure (request not received by the server) fires a change to the frequency of requests, which appears to be doubled (until the upper bound of 300000 milliseconds is not reached). The very first read operation on data happens at application launch, when the top level of directories is fetched and displayed: it is easy to observe, by turning connectivity off, that actually the synchronization mechanism fetches not only the current level, but also the subsequent one, guessing user navigation. This is a clear implementation of look-ahead technique. The application behaves differently with respect to data types: the analysis spans over media files and generic documents. Thumbnails associated to supported media file types are not cached directly when performing pre-fetching, as instead are downloaded following a lazy loading policy: more precisely, this is done as a response to scrolling event and only if the associated entry becomes visible in the UI. Cached thumbnails are stored in “/storage/Android/data/com.dropbox.android/cache/<id>/thumbs/”, embracing the standard storage directory structure in Android. It is worthy to analyze the behavior of the application when opening pictures: the original definition versions are cached together with the associated thumbnails, but applying look ahead policy. Matter of fact, when browsing images in directories comprising more than one media file, pictures are cached two at a time: the selected one, and the subsequent. We can conclude that this policy is datatype-specific. Touching documents entries triggers the download of an instantly server-side generated preview (Dropbox calls

this feature “Quick Preview”), a pdf file cached to “/storage/. . . /cache/<id>/ docpreviews/”: in this case, there is no look-ahead policy implemented.

Scenario 2: Posting (Data Created or Modified at Mobile-side)
Currently, two posting options are available: creating a new text document or uploading generic files. The former option opens the editor, whose behaves as follows: the application “watches” the open file to be notified of changes, which fire the upload task. The updated document is uploaded only after user interaction (via “Save” button), however this operation does not force the user to leave edit mode, so the file stays in “watch” state and the “updateIfChanged” flag is reset, waiting for the next “Save” action. Uploading a generic file is more straightforward: the upload operation is monitored using a progress bar. This process is likely asynchronous, since it does not appear to block the main thread of the application, so that the user is able to use the application while uploads are completed. Data deletion at mobile side leaves no evidence in the log; however, it should behave in a transactional way: note that, if the document to delete is cached, the corresponding file is deleted.

Scenario 3: Data Created or Modified via Backend Services
Even though long polling provides an update mechanism that is close to be instantaneous, the UI does not react immediately to changes to data performed via backend services. The only way to refresh the displayed elements is to trigger on-demand synchronization interacting with the “Refresh” button or to navigate back and forth to reload the browsing context.

Scenario 4: Data Conflicts Induction
Concurrent changes to documents are handled in a simple way: the file is duplicated and the most recent copy is labelled as “conflictual copy of <user name>”.

1.6 Case study 3: Instagram

1.6.1 Triggering Dimension

Synchronization happens as a response to some triggering interactions: launching the application, posting a picture or video, modifying or deleting a post and finally, forcing it by performing a specific associated gesture (i.e. pulling down the list of posts from its top entry and releasing it).

Automated synchronization is handled by push notifications, which are used to notify the user in various situations: however, their configuration is deeply customizable using the dedicated settings menu. By inspecting the add-ons used by the application, it appears that – as most of Android applications do – it adopt Google Cloud Messaging to send data from servers to client.

1.6.2 Storage and Transfer Dimensions

Storage strategy. Instagram[5] stores its entire cache in the system /data partition (path: "/data/com.instagram.android"): images, videos, and databases.

Data transfer. Transfer is differential and ruled by timestamps, as the nature of this kind of client would suggest; functionally, it is very similar to a feed reader. Actually, it is even simpler, because it does not feature real-time post streaming.

1.6.3 Special Features

History of changes. Instagram features a fragment of its UI containing a timeline of the notifications the user received. This is a considerable aspect in terms of synchronization because it adds some complexity to the implementation.

1.6.4 Behavioral analysis

Scenario 1: Data Read at Mobile-side
Launching the application triggers the first synchronization event, whose goal is to load the main feed page: since the log does not expose anything, let us analyze the behavior of Instagram in terms of file system operations. When loading the feed for the first time, by default the synchronization process caches the media of exactly 9 posts: after that, a lazy loading policy is adopted to load the rest of the stream in response to user interaction. More precisely, as the user scrolls down,

[5]published by Instagram, version 6.17.1, tested on Android 5.0.2

for each post she /he exceeds with the gesture, the most recent post not loaded in the list is pre-fetched: this kind of strategy allows the total elimination of latency, making synchronization practically invisible and enhancing the user experience. Thanks to this observation, it is even possible to estimate the size of Instragam's cache directory, given by the formula "cache.size = 9 + position", where to the head of the list (the topmost post) is at position 0. However, when connectivity is not available, Instagram shows only a few posts with complete metadata and comments, even though cache contains more files: the explanation is given by the fact that the application uses a file in json format to build the main feed. This file ("MainFeed.json.0001" in the case analyzed) is stored in cache and continuously modified as the user interacts with the UI: it even contains all metadata associated to the posts, as well as comments (evidence of the fact that Instagram does not store them on mobile devices, preferring to load needed data on-demand). Code inspection would be needed to determine how frequently this file is modified (and the size threshold triggering its partial deletion), but this overshoots the goals of this behavioral analysis.

Scenario 2: Posting (Data Created or Modified at Mobile-side)
Posting on Instagram is a multi-step process, in which most of the tasks are executed at mobile-side: picking a picture or a video, apply adjustments and /or filters, set a description and even tags followed users (so there must be a database table locally listing them). Of course, the upload task requires connectivity and it is performed most likely in asynchronous way: a fixed unobtrusive dialog is pinned on top of the feed while the upload is performed, and it is dismissed at completion. Let us observe that, by default, the application saves original photos and videos on the internal storage right after publication, but this feature can be turned off.

Scenario 3: Data Created or Modified via Backend Services
The application notifies the user in various scenarios: likes to owned post, comments to owned posts, new followers, follow requests approvals, Facebook friends opening an Instagram account, Instagram direct requests, Instagram direct activities, tags received, reminders of unread notifications, first post of a friend. By default, all of the aforementioned events will trigger Google Cloud Messaging to send a push notification, having two effects: a system notification is displayed and the application's UI is decorated with notification bubbles comprising counters.

Scenario 4: Data Conflicts Induction
Due to the architectural design of Instagram, currently it is impossible to perform or induce conflictual operations: for instance, the web client does not provide the possibility to edit or remove an owned post.

1.7 Analysis Report – Tabular View

Table 1.1: Triggering Dimension

	Evernote	Dropbox	Instagram
Automated periodic synchronization	✓		
Automated instantaneous synchronization		✓	✓
Interaction-based synchronization	✓	✓	✓
Lazy loading of data	✓	✓	✓
Forced on demand synchronization	✓	✓	✓
Gesture-based synchronization			✓

Table 1.2: Storage and Data Transfer Dimensions

	Evernote	Dropbox	Instagram
Storage of the entire dataset		✓	
Storage of a subset of the dataset	✓	✓	✓
Transfer of all needed data at once			
Differential transfer	✓	✓	✓
Transfer based on datatype		✓	
Look-ahead policy over transfers		✓	✓
Conflict resolution	✓	✓	

Table 1.3: Special Features

	Evernote	Dropbox	Instagram
Offline mode	✓		
Immediate feedbacks	✓		
History of changes			✓
In-app notifications			✓
Clean/dirty flags	✓		

Chapter 2

User Interaction Patterns

2.1 The Interactional Perspective

When speaking about mobile applications, user experience – and, more in general, front-end design – has a central role. As extensively reported in [7], "A successful user experience is crucial for successful application adoption". User experience and synchronization mechanisms are closer than we would think in the first place in the mobile environment, since they are hand in glove with performance, which is a major parameter in determining the optimality of a well-thought software. Sometimes the combination of good user interface and satisfactory performance leads to a successful commercial outcome regardless of the originality of the idea inspiring the application; in fact, users tend to prefer applications that look fancy and reactive, even if they are not the best when it comes to fulfil their necessities. To produce such positive outcome, user-centered design is a sensible option to serve development of mobile applications. As stated in [9], for mobile devices this means users need to feel a continuous thread between the interactors they are maneuvering and the software working. Furthermore, the experience must be delightful, possibly rich and uninterrupted: animated transitions, uncluttered and minimalistic menus and visual elements are frequently cited in design guidelines of the main mobile operating systems. Depending on the functionalities and goals of the application, data-flow events may play a crucial role in terms of interactional limitations: for this reason, even when designing the synchronization logic, considering its correlation with the front-end as a fundamental factor may represent a good advice.

In this chapter, we combine the above considerations over interaction with the triggering dimension of synchronization processes analyzed in the previous chapter, aiming at the definition some interactional patterns. Later in our study, we will link these patterns to other ones modeling the core of synchronization processes to build exhaustive models.

2.2 Interaction Flow Modeling Language

As we have been stating so far, we want to express the patterns we are going to present in the most abstract way possible, to preserve the platform-independence of our study. Under this perspective, the Interaction Flow Modeling Language (IFML) —adopted by the Object Management Group (OMG) as a standard in July 2014— appears to be an excellent technology, being a modeling language that "supports the specification of the front-end of applications, independently of the technological details of their realization" [4]. In fact, IFML addresses several questions related to front-end modeling, such as view composition and content, commands, actions, effects of interaction and parameter binding. IFML syntax is visual, based on OMG standards and very close to the one characterizing Unified Modeling Language, thus it results familiar to developers. Its expressive power is empowered by extensibility, supported by the language thanks to its incorporated standard means for defining new concepts. Practically, this quality is being exploited to propose a model-driven approach for Mobile Application development, as portrayed in [6] and witnessed by innovative commercial solutions like WebRatio Mobile Platform [2]. Considering all of the abovementioned advantages, the use of IFML to satisfy our inquiry is a logical consequence. In the next section, we are introducing the patterns depicting the front-end implications of data synchronization triggering using the IFML notation, fully referenced in [4]. More details on IFML are available in the Related Work chapter.

2.3 Patterns Identification and Analysis

Pattern Form. Pattern are illustrated in compliance with the Coplien Form by James O. Coplien, "one of the founders of the Software Pattern discipline, a pioneer in practical object-oriented design in the early 1990s and is a widely consulted authority, author, and trainer in the areas of software design and organizational improvements" [1]. We slightly alter the pattern form by adding a visual explanation expressed in IFML, so the resulting structure is arranged as follows:

Problem
Statement of the problem(s) that the pattern works to solve.

Context
Definition and constraints of the specific context of applicability of the pattern.

Forces
Scenarios and motivations supporting the pattern logic or application.

Visual Explanation
A diagram or graphical representation explaining the pattern and a description of the interaction between the different elements of the pattern.

Solution
Explanation of how the pattern solves the problem considering the constraints of the context.

Resulting Context
Definition and constraints of the context resulting from the application of the pattern.

Rationale
Philosophy inspiring the pattern.

2.3.1 Pattern: Content Scrolling

Problem

Lazy loading is a requirement to preserve performance when scrolling long lists. When data in the list has to be synchronized, the problem becomes more complex, and the event must fire without interrupting the interaction.

Context

Browsing a list – or, more generically, a scrollable container – comprising contents to be synchronized.

Forces

- When browsing lists containing a notable amount of elements to be loaded with a lazy policy, synchronization events are inevitably very frequent
- Requesting additional input to the user to perform synchronization annoys her/him and worsens the user experience
- In the same way, alerting dialogues and obtrusive feedbacks distract the user from its goal (i.e. exploring, browsing)
- The user does not expect the synchronization event to be needed

Visual Explanation

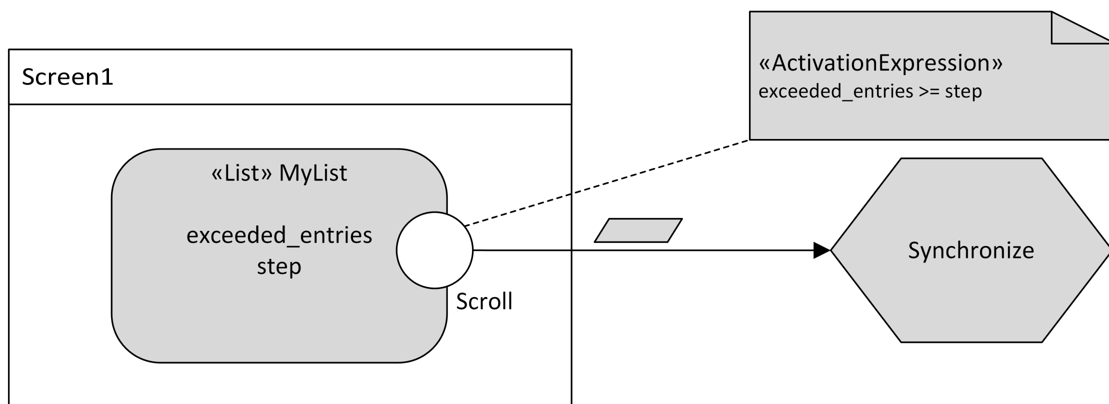


Figure 2.1: Content Scrolling pattern rendition in IFML

Solution
Triggering a synchronization process in a very transparent way, seamlessly and without requiring a dedicated visual element to interact with. The event fires when reaching a certain relative position in the list with respect to an entry (represented by the attribute *exceeded_entries* in fig. 2.1 on the facing page). For convenience, we are naming *step* the amount of entries to be exceeded.

Resulting Context
The context remains unaltered, as the user would expect it. New content is loaded and displayed.

Rationale
Loading content is a responsibility of the application components: users should not be requested to perform ad-hoc actions, so this pattern exploit the natural interaction as a trigger.

2.3.2 Pattern: Context Change

Problem
In some applications, the User Interface is layered, with tree-structured contents. Exploration is required, and content refresh has to be performed at each step.

Context
The context resembles a hypertext: interacting with elements in the container mimics the behavior of navigation links in a web page.

Forces

- Load all the data at once can be rather inconvenient when the application does not need to display all of its contents in a single view
- Exploring tree structures implies the frequent invalidation of the on-screen content
- Displaying obtrusive elements to start synchronization events interrupts the user experience
- Exploiting transition times to load content makes the experience continuous and smooth

Visual Explanation

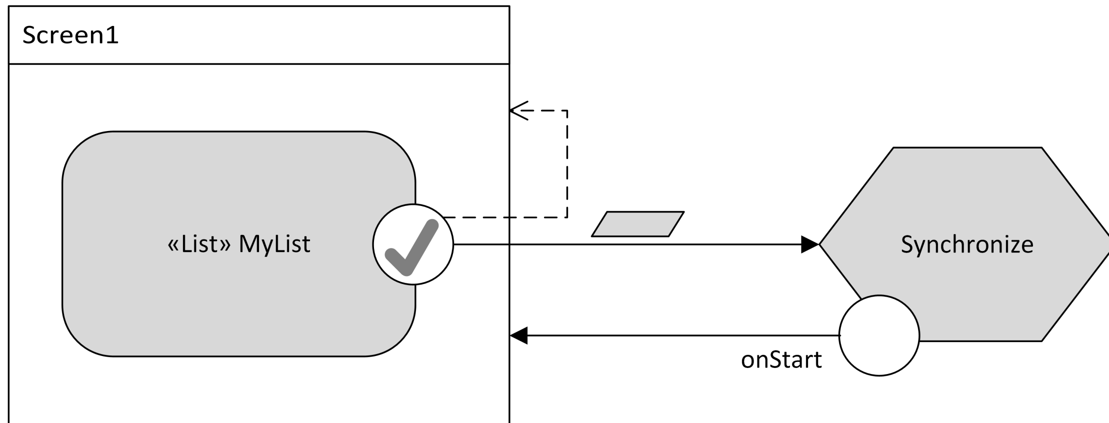


Figure 2.2: Context Change pattern rendition in IFML

Solution

Selecting an entry within a nested structure triggers the data alignment event and the subsequent refresh of the entire content displayed. Information on the change is passed to the container, as represented by the data flow connecting the selection event to the screen in fig. 2.2. As the action starts (event *onStart*), the focus is given back to the screen displaying the list.

Resulting Context

The frame does not change and, ideally, the interaction pattern remains unaltered and applied to the new content.

Rationale

Applications adopting the "explorer" paradigm often have to deal with large amounts of data. If synchronization is involved, then downloading everything at in a single transfer can be critical in terms of performance and bandwidth. Synchronizing the minimum amount of needed data at each step may be eased by the application of the context change interactional pattern.

2.3.3 Pattern: Pull-To-Refresh

Problem

On-demand synchronization has to be featured as an option, but classical implementations like buttons in menu-bars look outdated and disrupt the user experience.

Context

Usually a scrollable view container, but it can be a generic one.

Forces

- This pattern is becoming a standard for synchronized lists (e.g. timelines) and, more in general, scrollable components
- A vertical gesture requires less effort compared to the selection of a traditional entry in a menu, given the morphology of most of mobile devices: as a result, the user experience is more continuous
- This pattern can replace the "refresh button" in user interfaces, helping to keep them minimal

Visual Explanation

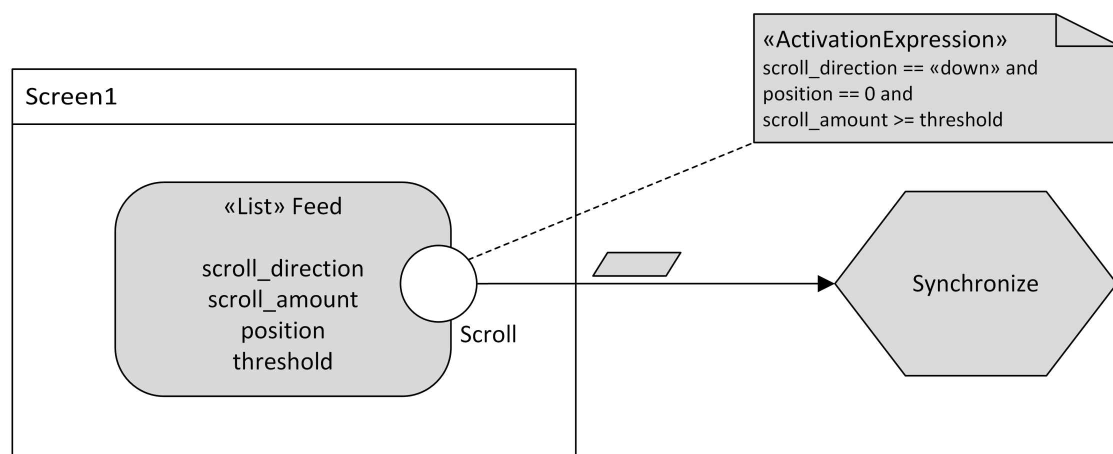


Figure 2.3: Pull-To-Refresh pattern rendition in IFML

Solution
The functionality is straightforward: the synchronization process is triggered by the completion of the gestural combination. Specifically, given that the user is pointing at the first element of the list, a fling down gesture makes the list "draggable" from the top to the bottom; to complete the action, the list must be released after having been dragged for a certain vertical distance(*scroll_amount*): we have named *threshold* the minimum distance to exceed to make it functional. These conditions are collapsed in an *ActivationExpression* associated to the *Scroll* event in fig. 2.3 on the preceding page.

Resulting Context
The pattern makes the target view container "pullable".

Rationale
This pattern combines some key elements of user interaction in mobile devices and applications and offers a straightforward solution to keep the experience as natural as possible. As a downside, affordance of the gesture-outcome pair is not very intuitive, but its recall is continuously growing as many popular applications are adopting it.

2.3.4 Pattern: Form Submission

Problem
After filling a generic form, a synchronization event has to be started, so the interactor must be unambiguous and recognizable.

Context
The view comprises a form and a submit interactor, whose layout depends on the implementation variant.

Forces

- When user input is required, the most serviceable way to register it is to commit it after manual confirmation

Visual Explanation

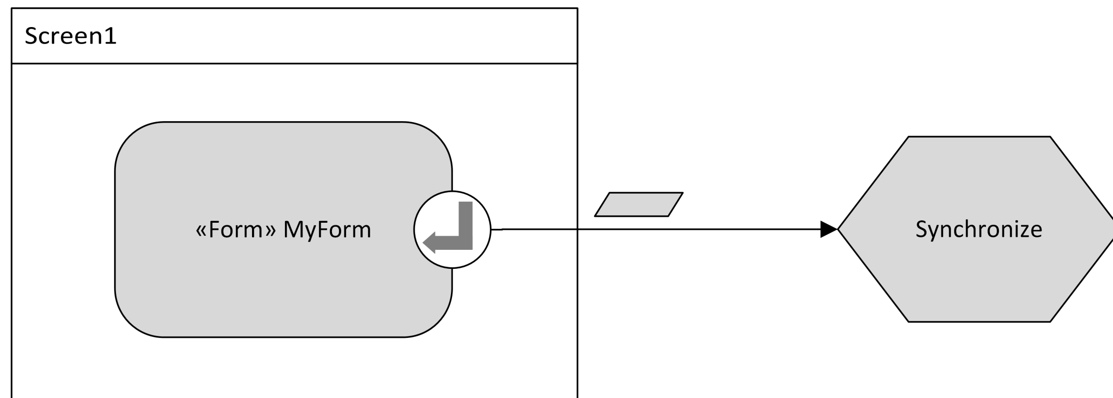


Figure 2.4: Form submission pattern rendition in IFML

Solution
Keeping the interaction mode as simple as possible to trigger a synchronization process.Typically it relies on a simple touch on a self-explanatory submit button. A *ParameterBinding* handles the dependencies needed to perform the action, as shown in fig. 2.4.

Resulting Context
The resulting context may vary according to the implementation logic. For instance, it can comprise a modal pop-up or a visual element pinned to the parent view signaling the operations in progress.

Rationale
Committing an operation requires the complete consciousness of the user, whose first concern is typically about performing a wrong action on data. The easiest way to letting her/him know of the synchronization process is indeed the application of this pattern.

2.3.5 Pattern: Application Launch

Problem
Data alignment needs to be performed to achieve the basic functionality of the application.

Context
Application is launching its landing activity and loading the corresponding view.

Forces

- Having up to date content displayed at application launch improves the perception over user experience
- Occasionally some functionalities are inhibited without fresh data, so – if it is the case – the soon it is obtained, the better the application works
- Having data aligned at application launch gives the user the opportunity not to refresh manually

Visual Explanation

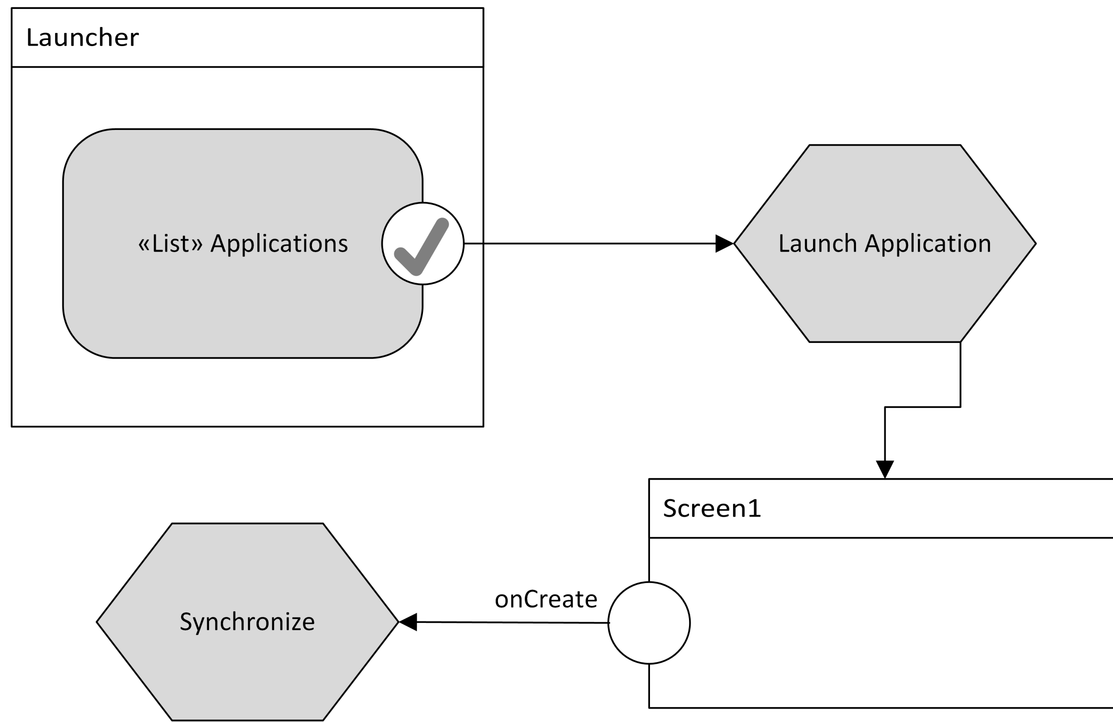


Figure 2.5: Application launch pattern rendition in IFML

Solution
Exploiting the application launch event to trigger the synchronization process start. As all the tasks are completed, depending on the implementation, the resulting view is refreshed or populated for the first time. To emphasize the action starting as the screen is created, in fig. 2.5 we put an *onCreate* event on the boundary of the screen element.

Resulting Context
Main activity context of the application, populated with up to date content. Rationale

Rationale
If the application is based on data that needs to synchronized, performing alignment on start is a good way to improve the overall experience and let the user out of the underlying logic.

Chapter 3

Data Synchronization Logic

In the previous chapter, we have been discussing the interactional aspects of the triggering dimension of data synchronization. As our studies are centered on mobile applications, this perspective is certainly one of the most interesting and dense of consequences over the whole modeling process. On the other hand, we inevitably have to deal with other technical aspects characterizing the core phase of data alignment mechanisms.

In this chapter, we want to portrait the possible choices in terms of implementation when designing the alignment logic between two data stores. In particular, we will focus on the aspect of communication, to emphasize the "directional" dimension of synchronization. Finally, this reasoning will lead us across some reflections about the rise of conflicts and their resolution techniques.

Eventually, we will focus on Client-Server communication to review some already analyzed dimensions of data synchronization under a different perspective, bound to the aspects discussed in this chapter. In this way, we will understand how architectural choices, goals and requirements affect data alignment policies in mobile applications. This in-depth analysis will help us in understanding how to integrate front-end modeling with data synchronization design, which takes place between front-end and back-end.

3.1 Synchronization Types and Elements

In this section we clarify how the data synchronization problem is faced at different levels, eventually underlining the inner complexity introduced by its elements. All of the considerations reported below refer to [10] by Drew McCormack.

3.1.1 The Synchronization Grid

We start by categorize data synchronization technologies according to their most characterizing features: synchronicity and "end" profile. The resulting grid is featured in [10] and is thought to collocate all existing technologies according to the aforementioned parameters.

	Synchronous	Asynchronous
Client-Server	Parse StackMob Windows Azure Mobile Services Helios Custom Web Service	Dropbox Datastore TouchDB Wasabi Sync Zumero
Peer-to-Peer	iTunes/iPod Palm Pilot	Core Data with iCloud TICoreDataSync Core Data Ensembles

Table 3.1: Grid classifying the existing synchronization technologies

Data alignment uses Synchronous Communication (SC) when each request by the client (or peer) expects to receive a response in real-time. Asynchronous Communication (AC) instead provides a framework acting as a proxy between client and server (peer and peer), where data transfers are not bound to real-time responses; at contrary, they are performed in the background. Orthogonally with respect to synchronicity, the Client-Server communication usually provides a "smart" end encapsulating most of the synchronization logic, while with Peer-to-Peer the client apps handle all of the complexities. In terms of trends, we are moving towards Asynchronous Peer-to-Peer, which is easily the most complex combination in terms of architectural design, but also the most interesting thanks to the architectural-independence it provides to mobile application developers.

It is important to underline, quoting [10], that “history of sync does not follow a single linear path. Each stage overlaps with those that follow, continuing to be utilized even as new approaches evolve. Today, all of these techniques still exist and are in active use, and each may be an appropriate solution to your particular problem”.

Let us now picture the main characteristics of each of the approaches resulting from previous the tabular synthesis, to give an idea of their logic and possible implementations. The following overviews are summarizing the in-depth reports presented in [10].

Synchronous Communication, Client-Server. This is probably the most common approach to synchronization in use today. From an implementation standpoint, it recalls web services: in fact, typically, a custom cloud app is developed in a language, and with a programming stack unrelated to the client app, such as Ruby on Rails, Django, or Node.js. Communication with the cloud is slower than using a local network, but of course, S-CS has the advantage of being “always on” so that the client can synchronize from any location, if network connectivity is available. Costs are evaluated in terms of data transfer and storage, consequently we need to reduce the communications overhead as much as possible.
The solution is given by differential transfers based on change tracking: one device sends changes to the other, and the receiver merges and sends back a set of changes incorporating the results of the merge.

Synchronous Communication, Peer-to-Peer. This approach was actually the first to be broadly adopted, and used for peripheral devices like iPods and PDAs. S-P2P tends to be simpler to implement, and exploits the fast connections of local networks.
A synchronization operation involves one device transferring its store to another device, which determines what has changed, merges, and sends back the resulting store. This flow scheme guarantees that both devices have the same data after a synchronization process, with great robustness.
An example is given by the implementations of iTunes, still based on this approach due to the large quantities of media transfer involved.

Asynchronous Communication, Client-Server. With A-CS, the developer adopts an API for data storage, which gives access to a local copy of the data. Synchronization occurs transparently in the background, with the application code being informed of changes via a callback mechanism. Examples of this approach include the Dropbox Datastore API, and – for Core Data developers – the Wasabi

Sync service.
The main difference between A-CS and S-CS is that the extra layer of abstraction provided by the framework in A-CS shields the client code from direct involvement in alignment logic. It also means that the same service can be used for all data models, not just one particular model.

Asynchronous Communication, Peer-to-Peer. A-P2P places the full burden of alignment logic on the client app, without any recourse to direct communication. As with A-CS, each device has a full copy of the data store. The stores are kept synchronized by communicating changes between devices via a series of files, typically referred to as transaction logs. The logs are moved to the cloud, and from there to other devices by a basic file handling server (e.g. iCloud, Dropbox), which has no insight into the file content.
A-P2P is challenging to implement, since it has to deal with several complexities, the most critical of which are the lack of "truth" convergence and subsequent tricky change tracking (due to the idiosyncrasies related to timestamps and out-of-order operation processing).

3.1.2 Change Tracking

Change Tracking is the practice of registering within the synchronization algorithm what properties of data have changed since latest alignment event, aiming at determining what should be altered in the local store. Note that every change affecting and object is generally handled as a CRUD operation and labelled as a "delta" (term emphasizing the differential nature).

The granularity of change tracking is one of the first parameter to define when designing: in fact, according to our requirements or constraints we should determine whether it might be useful to update all properties in an entity as a response to a single property change. Otherwise, recording the individual changed property is the choice, to reduce overhead at the cost of raising complexity.

The need for a mean to record changes is unquestioned, anyway. It may be just a Boolean attribute in the local store to indicate whether the object is new or has been updated since the last alignment. Adding some sophistication, changes could be recorded outside the main store as a dictionary of changed properties with an associated timestamp.

3.1.3 Conflict Resolution

The potential for conflicts exists on every occasion we are dealing with two or more stores representing the same logical set of data. A change to an object in one

store could fire at about the same time as a change to the corresponding object in a second store, without intervening synchronization. These changes are labelled as concurrent, and some action may be necessary to leave the conflicting objects in a consistent and valid state across all stores once they are aligned.

Simplifying this scenario as much as possible, reading and writing a store can be considered an atomic operation and resolving conflicts purely means choosing which version of the store to keep – this is actually more common than we might think.
Otherwise, changes precedence when resolving conflicts can be thought according to multiple criteria. When a central server is involved, the most straightforward approach is just to assume the latest synchronization operation takes priority. Any change present in the operation overwrites previously stored values. With a more complex procedure, change precedence is determined by means of the comparison of creation timestamps of conflicting operations.
Although approaches to conflict resolution may vary, it is critical that the resolution stays deterministic. This means that if the same scenario occurs on two different devices, they should end up taking the same action.

Conceiving a model that cannot become invalid because of concurrent changes, if possible, is the best choice, since it avoids us the complexity brought by conflict resolution. Of course, this is much easier when starting a new project, because reasoning on potential invalid states may be a difficult, time-consuming task.

3.2 Client-Server Communication

Across the previous section, we distinguished architectures built upon the client-server communication schema from the ones in which each host (peer) has the same functional role with respect to data synchronization.
In this section, we concentrate on the former, scrutinizing communication between client and server to better understand how it may shape according to functional and non-functional software requirements.

3.2.1 Directional Aspects

When referring to C-S synchronization so far, we have been focused on the synchronicity of the process and on the differential nature of transfers. Nevertheless, like any other communication, it features a directional property.
By inspecting the Data Synchronization API of Tizen (convenient to analyze since it exposes its functionality very clearly), mobile OS based on the Linux kernel and governed, among the others, by Samsung and Intel, we can observe how this property defines the roles of sender and recipient and the reciprocity of the messages. Below we report the possibilities the aforementioned API provides.

Two Way. Indicates a symmetric synchronization type in which the client and the server exchange information about modified data in these devices.

One Way from Client. Indicates a synchronization type in which the client sends its modifications to the server, but the server does not send its modifications back to the client.

Refresh from Client. Indicates a synchronization type in which the client sends all of its data from its store to the server (backup). The server is then expected to replace all data in the target store with the data received from the client.

One Way from Server. Indicates a synchronization type in which the client gets all modifications from the server, but the client does not send its modifications to the server.

Refresh from Server. Indicates a synchronization type in which the server sends all its data from a store to the client (restore). The client is then expected to replace all data in the target store with the data received from the server.

3.2.2 Temporal Aspects

Periodic Synchronization

Running data synchronization as a scheduled task may represent a fundamental feature. The idea is to force the process start every certain time interval, which, optionally, may be configurable by the user.
The key software requirements leading to the implementation of periodic synchronization are the high need for fresh data, content refresh in background and, more in general, automation. In terms of non-functional requirements, the User Experience benefits from the independence of the process from user input. Furthermore, if configuration is provided, it increases the user's possibilities. Finally, a smart implementation, maybe in coordination with the back-end services, can positively affect battery and mobile data consumption.
Speaking of applicability, periodic synchronization well fits the needs of informative, data-intensive applications, especially if featuring dynamic widgets. Among the examples, we can collocate weather or transportation forecasting applications and widgets, but also connected personal to-do lists and portfolios. Some impediments may be represented by OS-logic related environmental conflicts, which arise from various conditions, such as battery level, connectivity or system preferences.
From a technical point of view, operational asynchronicity is almost a requirement due to the background processing needs of this implementation.
As already mentioned when introducing client-server communication, differential transfer is the preferred choice to reduce overhead, being able to leverage both communication and processing.
A good example of periodic synchronization, as underlined in Chapter 1, is given by Evernote's data alignment policy, which also provides configuration settings.

On-Demand Synchronization

The concept of on-demand process start is very common when it comes to synchronization events. It is even mandatory when data alignment is a component in a larger business process.
The idea is to bind the triggering of the synchronizing action to some specific events and conditions (in the typical Event-Condition-Action scheme). Adopting on-demand synchronization, the front-end designer is able to set the opacity of the process according to non-functional requirements, deciding if the user should be acknowledged of triggering or running state of data alignment. On-demand data synchronization is essential when aiming at realize a responsive UI in applications based on layered-structured content to be browsed, like clients for cloud-based web applications.
In opposition to periodic synchronization, this temporal scheme provides that most of the complexity is handled by the client, in a way that queries respond to the

restrictions dictated by the application context (obviously unknown to the server). In accordance with this hypothesis, transfers are often differential with a dual purpose: exploiting the client-side complexity level and keeping the experience responsive and continuous thanks to negligible processing effort requested.
Privileging on-demand synchronization over other temporal scheme may provide better battery, bandwidth and mobile data consumption. As a downside, if it is designed to be the only scheme to be implemented, its failure becomes slightly more critical, as obsolescence rate of data is increased with respect to periodic refreshing, for instance.
In conclusion, on-demand synchronization is widely used in mobile applications, being almost necessary to provide a decent user experience. In fact, in our investigation reported in Chapter 1, we identified it in all the applications under inspection.

Instantaneous Synchronization
The concept of instantaneous synchronization is the result of the merge of the ideas of notifying the client of changes and aligning data.
The adoption of this temporal scheme must meet the functional requirements, among which the immediate update of contents with the minimum delay possible must be a priority. Otherwise, the need for notifications falls, and with it, the idea of bind them to a synchronization event.
The typical approach to this implementation involves push notifications (i.e. messages sent from the server to a push notifications provider, which dispatches them to the client application), but it can be used with pull mechanisms as well (less efficient in terms of computational resources used and battery consumption).
Data synchronization is triggered by the notification messages, so the complexity is balanced on client and server, while transfer are mostly differential, thanks to the information sent by the server.
Instantaneous synchronization is the choice when mobile applications have to notify the user about important changes, like social interactions, critical updates and reminders.
As an example, we can cite the notifying implementation observed in Instagram, as reported in Chapter 1.

Chapter 4

Data Synchronization Patterns

Synchronization Patterns. The core phase of data synchronization encapsulates the most complex and diverse logic of the entire process. As we have seen in the previous chapters, there are several factors to consider when trying building a model to describe data alignment. Furthermore, modeling the dynamic behavior adopted during the client-server communication would be not only impossible, but also pointless. What is interesting is instead abstracting the logic of the process under three specific dimensions: time, amount of data to store and sophistication of the transfer.

This three-headed inquiry helps us to identify some typical patterns that are already adopted and implemented in real-world applications, and not only within the mobile ecosystem. Actually, the abovementioned investigation has been successfully achieved by McCormick and Schmidt in [11], which features a valid set of patterns to analyze. As explained in the original study, most of the recognized patterns are architectural patterns, which means that they should be viewed as structural organization schemas addressing the problems of synchronization in a variety of different contexts.

Pattern Form and Grouping. In [11], data synchronization patterns are thoroughly classified using a variant of the *Gang of Four* and *POSA* pattern forms. In this chapter we cite and credit the aforementioned study, trying to improve patterns' visual representations and to classify them in a more compact depiction. We reviewed the pattern representation to adopt the Coplien Form already used to present User Interaction Patterns. We slightly altered the pattern form by adding the pattern's intent and a visual explanation expressed by UML diagrams, so the resulting structure is arranged as follows:

Intent
Statement of the intent of the pattern.

Problem
Statement of the problem(s) that the pattern works to solve.

Context
Definition and constraints of the specific context of applicability of the pattern.

Forces
Scenarios and motivations supporting the pattern logic or application.

Visual Explanation
Diagrams or graphical representations explaining the pattern and a description of the interaction between the different elements of the pattern.

Solution
Explanation of how the pattern solves the problem considering the constraints of the context.

Resulting Context
Definition and constraints of the context resulting from the application of the pattern.

Rationale
Philosophy inspiring the pattern.

Finally, to better classify patterns, we added a grouping logic based on the three characterizing dimensions we mentioned before: time, storage strategy and transfer logic sophistication.

4.1 Patterns By Time

4.1.1 Asynchronous Data Synchronization

Intent
Performing all tasks required by data synchronization in asynchronous way, allowing parallel on-screen operations. This means that the event does not block or consistently slow down the user interface.

Problem
Several connected mobile applications have quick access to data as a non-functional requirement. Speaking of rapidity in the mobile environment, in terms of UX responsiveness and waiting time are key elements. That been said, these two aspects are complementary and should ideally be balanced: a responsive interface taking ages to load data leads to user frustration, just like a quick data-loading interface lacking in responsiveness feels obsolete.

Context
While the current state of the application may be rather generic, the next state should not strictly depend on the result of data alignment event, especially when the pending operation includes uploads. This context matches all the scenarios in which the application features consumption possibilities and the user performs uploads or downloads on-demand, just occasionally. Clearly, the outcome of those tasks does not change the information consumption experience: social networks' clients are a fitting example. More broadly, applications in which stale data is valuable in terms of browsing because of highly informative contents or quasi-static data are ideal candidates: blocking the interface to obtain fresh data —not a mission critical operation— would not make sense.

Forces

- Synchronizing data while maintaining the application availability greatly improves the user experience: it is as valuable as an actual feature.
- Background synchronization is noticeably less obtrusive in terms of computation on multitasking-enabled systems: asynchronous synchronization is a best practice in this scenario.
- A non-immediate response may be commensurate to the need of fresh data, which is not always critical.

Visual Explanation

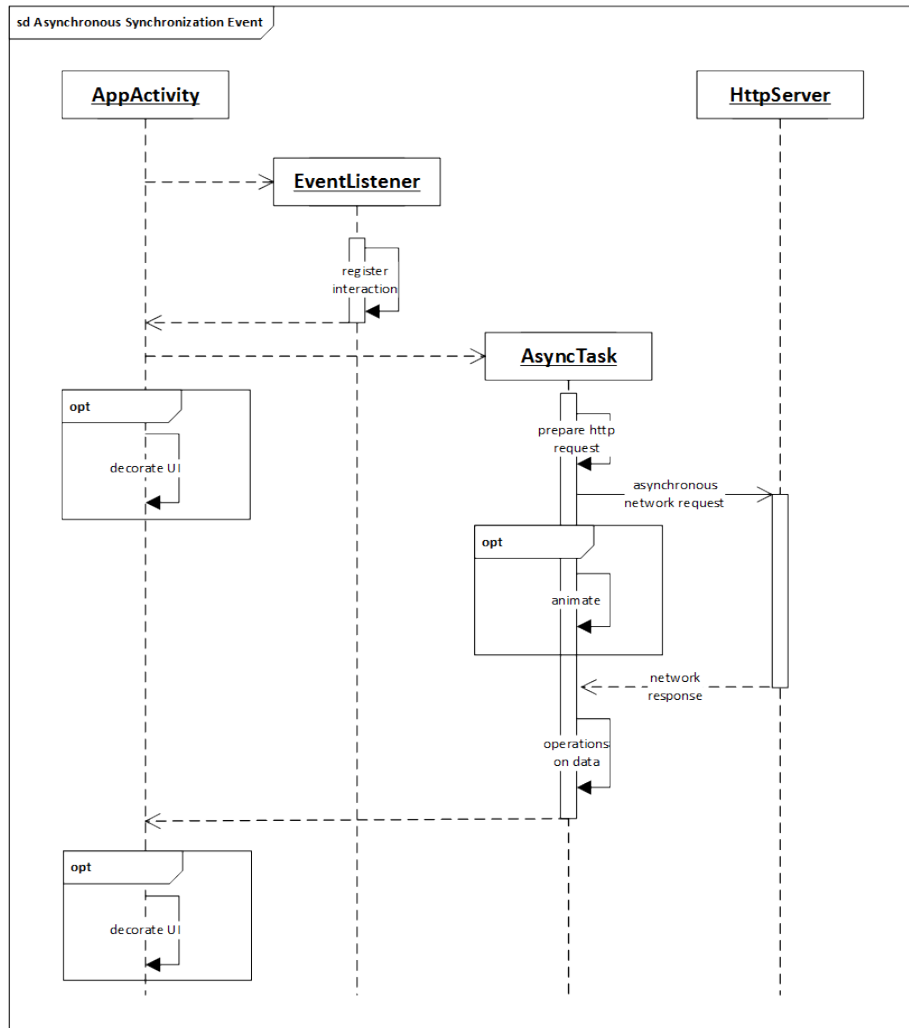


Figure 4.1: UML Sequence Diagram for Asynchronous Data Synchronization

Solution

The data alignment operation is triggered by a generic event, being it an aware interaction or a state change detection (refer to Android Intents or iOS push notifications). As shown in fig. 4.1, the application provides an *EventListener* capturing the event and communicating it to the context. As the delegate object is informed (in the diagram, for simplicity, we picked the same activity instantiating the listener), an asynchronous task implementation (*AsyncTask in the diagram*) is

instantiated. The lifecycle of this object is completely disconnected from the one of the application, in such a way not to interfere with its functionality. Optionally, an animation signaling the running alignment process may be handled by the task itself. At synchronization completion, a message is sent back to the calling object, or even broadcasted, depending on the implementation. Optionally, a feedback signaling the end of the operation may be displayed via notification mechanisms (for instance, Android *Toasts* or iOS *UIAlertViews*).

Resulting Context

The context is refreshed and features new content is appended to stale data. Depending on the design choices, the content may be added incrementally, substitute stale data or kept ready until a certain interaction is registered (especially in case of background synchronization). Optionally, the completion of data alignment may be notified within the interface of the application or exploiting a system-level implementation. A problematic resulting context may be generated when inconsistencies are raised from concurrent access to a shared dataset. Depending on the application's requirements, this may represent a major issue or a negligible flaw.

Rationale

Asynchronous synchronization is one of most commonly used data synchronization patterns in mobile applications. This is due to the fact that its advantages are evident, while drawbacks are proportionally less critical. In case of background synchronization the adoption of this pattern is even a requirement. The general improvement to the UI responsiveness that this pattern contributes to maintain favors its inclusion within the best practices for developers of many platforms.

4.1.2 Synchronous Data Synchronization

Intent
Performing data synchronization in synchronous way, making the application's components wait for its completion before starting any other operation.

Problem
In some scenarios, the representation of data within the application's interface must be extremely accurate, even close to a real-time mirroring. Additionally, working with stale datasets may result useless, hindering or eliminating productivity features of applications. More generally, sometimes the next state in a navigation session within the application may depend on the result of operations on data involving client-server or peer-to-peer exchanges, and the application of this pattern is an effective way to prevent the application from entering a non-functional state.

Context
Typically the application context features a state providing interactors to start a transition whose outcome is crucial to determine the subsequent state. Stale data is rather useless, since its reliability in terms of freshness and consistency is a fundamental requirement. Even out-of-order data transmission could represent an issue.

Forces

- Synchronicity is required when the application allows sensitive data modification via online services.
- Synchronous alignment resembles a transaction-like procedure, featuring benefits like atomicity, consistency, isolation and durability.
- Stale data may be useless or even harmful with respect to the application's functionality.
- Authentication-like scenarios are not functional before their completion.
- Synchronous operations may require less time to be completed due to their UI-blocking approach.

Visual Explanation

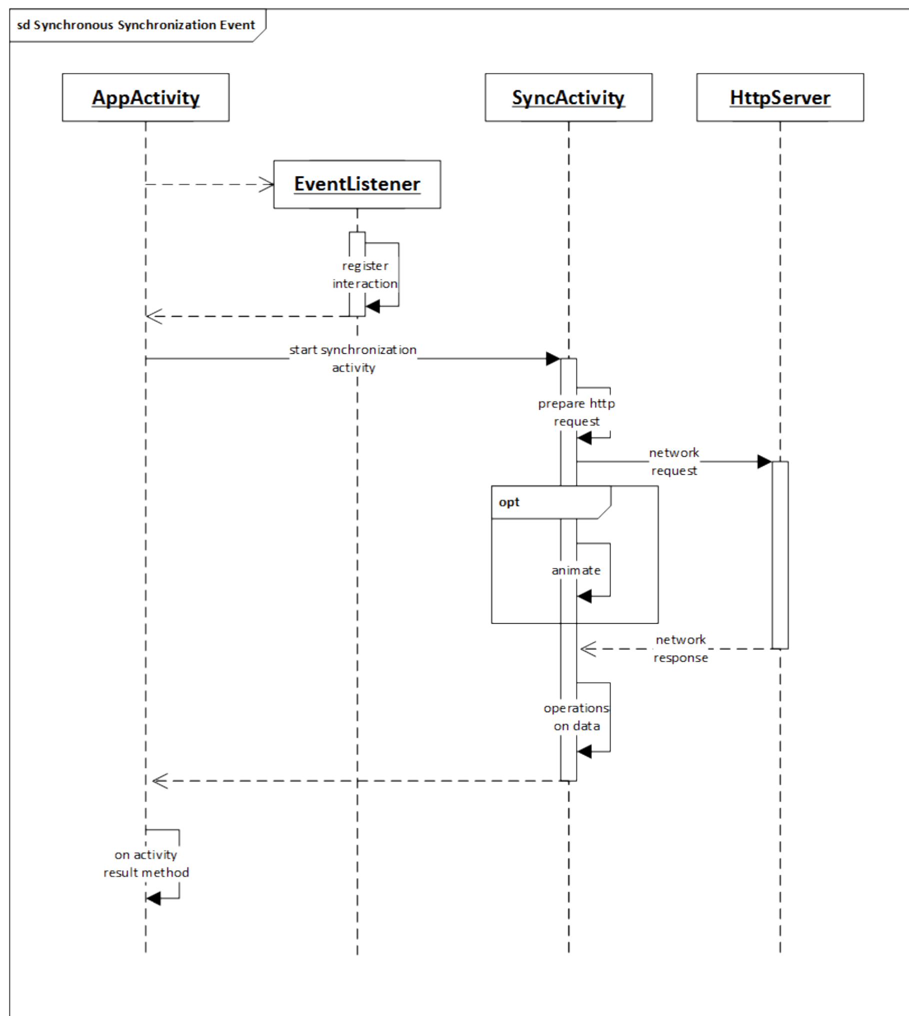


Figure 4.2: UML Sequence Diagram for Synchronous Data Synchronization

Solution

The data alignment process initiation is triggered by a generic event detection by a listener instantiated within the application (*EventListener* in fig. 4.2). The actual synchronization is not handled by an asynchronous or background task. Rather, it is managed by a dedicated activity (*SyncActivity* in the diagram) started with this purpose and running in foreground (this flow is represented by the synchronous message notation for *start synchronization activity* in the diagram), optionally starting a continuous animation until completion. This activity usually does not

provide interactors offering some functionality, except for contingent canceling options. As the previous activity is resumed, it performs some operations as a result of the termination of the data alignment process. In the diagram, this concept is pictured as a method call, named *on activity result method*.

An alternative simpler solution may provide a single foreground activity handling the whole synchronization process within its lifecycle, keeping the UI blocked.

Resulting Context

The UI is blocked, inhibiting the functionality of the application until the process' completion. This context may be problematic as any interruption, even if important (think to a network call, for instance), could interfere with the data alignment operation, with potentially weak results for both the interrupting task and the foreground synchronization. As synchronization is completed, the context provides a new state in which fresh data can be displayed according to the application's needs. This data is coherent with the order of the states that the application has undergone.

Rationale

The philosophy behind this pattern is to serve fresh data as a crucial component for the application to provide a certain functionality. We can even state that, by applying this pattern, serving fresh data becomes a functionality of the application. As mobile systems evolve towards multitasking scenarios, enabled by architectures supporting advanced parallelism, the adoption of this pattern appears to be discouraged. Actually, synchronicity may still represent the only effective solution for situations in which the data flow must stay coherent and predictable.

4.2 Patterns By Storage Strategy

4.2.1 Partial Storage

Intent
On the sidelines of a synchronization process, storing data on the device only as actually needed, to optimize network bandwidth and storage requirements.

Problem
Many applications work with large amounts of data to synchronize. Download operations obviously imply the storage of data on the devices' memory, but the their storage capability is limited and not intended to be saturated due to this mechanisms. This problem of storage is accentuated for applications that were originally intended to run on desktop systems, in which resources like network bandwidth and storage itself have always been more abundant.

Context
The fruition of contents provided by the application requires some expensive operations in terms of bandwidth and storage. The target dataset is too large to be entirely stored on the device, besides the fact that in most of the cases it is not needed.

Forces

- Devices' mass storage should remain available for other purposes.
- Unneeded data should not be stored.
- Large-sized resources should not be stored as well, especially if not consulted continuously.

Visual Explanation

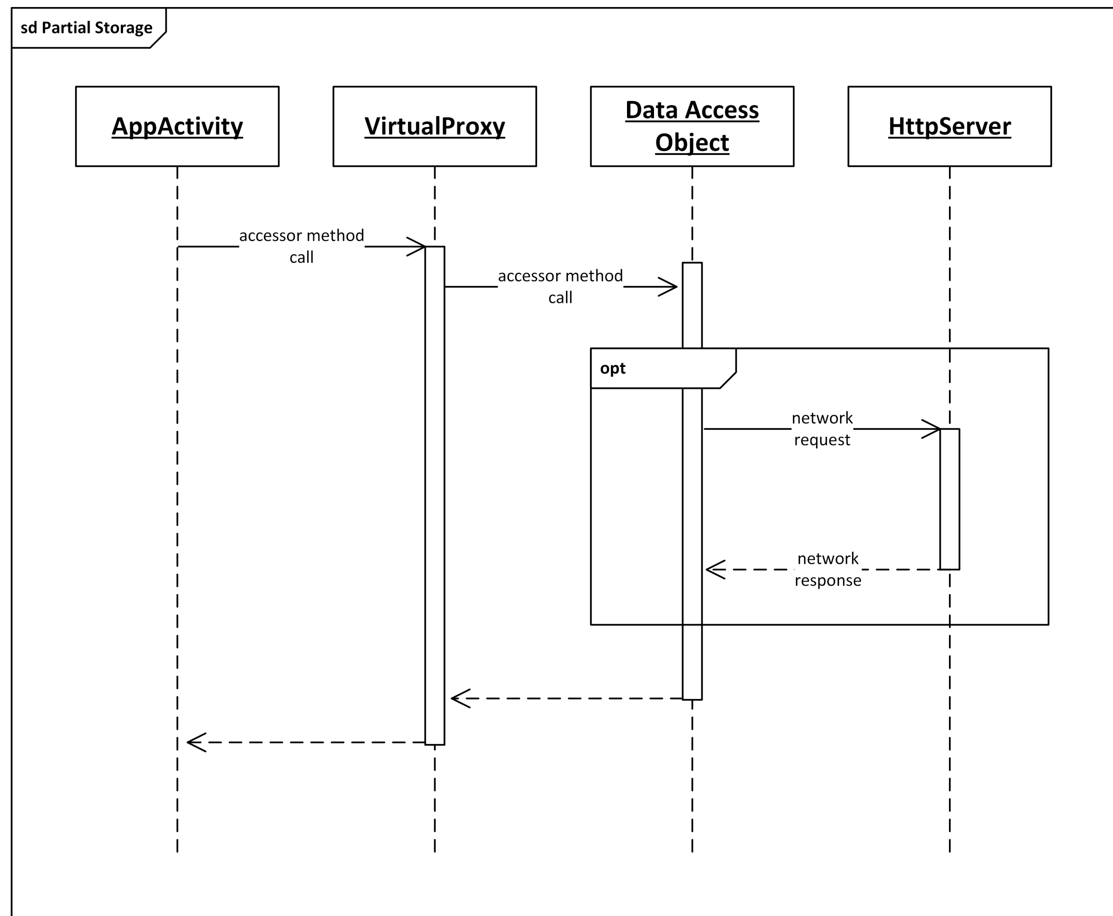


Figure 4.3: UML Sequence Diagram for Partial Storage

Solution

Instead of opting for a brutal pre-fetching, data synchronization is dynamic, on-demand, and triggered by some generic listeners in the application, most typically using a variant of the *Virtual Proxy* pattern. A virtual proxy is an object having the same interface as system objects intercepting certain events (for example method calls, short text messages, push notifications), in this case dedicated to handle the synchronization logic. The Partial Storage pattern can be realized using the Virtual Proxy combined with Data Access Objects (DAOs), having the virtual proxy triggering alignment when needed data is not in local storage (as determined by the DAOs). The virtual proxy instance queries the DAOs for the data to

populate its fields. Logically, as the proxy object is destroyed, its encapsulated data becomes unavailable.

Resulting Context
Fresh data is obtained incrementally and resides in cache until needed. Without network connectivity and with application components not loaded in memory, contents to be downloaded on-demand exploiting this pattern cannot be displayed.

Rationale
In a context in which mobile devices are designed to host large amounts of applications, the need for persistence of data must be carefully evaluated. Partial Storage provides a solution to filter data not to store physically, featuring an interface to handle on-demand transfers.

4.2.2 Complete Storage

Intent
Storing all the data to be synchronized on the device, so that is persistently available for consultation, improving responsiveness and nullifying loading times.

Problem
In all scenarios characterized by the absence of network connectivity, the consumption of online data is inhibited, as well as associated productivity features. Having data stored on the device's internal memory allows offline consultation and work. In fact, some applications meant to broadcast up-to-date emergency information, regarding for instance first-aid stations or shelters in case of natural disaster, are precisely designed to make data always available, due to the likely lack of connectivity implied by such a scenario.

Context
The context is usually identified with components of the application conveying information extracted by remote data. Typically, but not necessarily, the operations implied by Complete Storage are performed at application launch. However, the next state of the application requires data synchronization.

Forces

- Complete Storage enables offline mode.
- Low network bandwidth may negatively affect frequent data transfers implied by Partial Storage.
- The interface is considerably faster at loading contents.

Visual Explanation

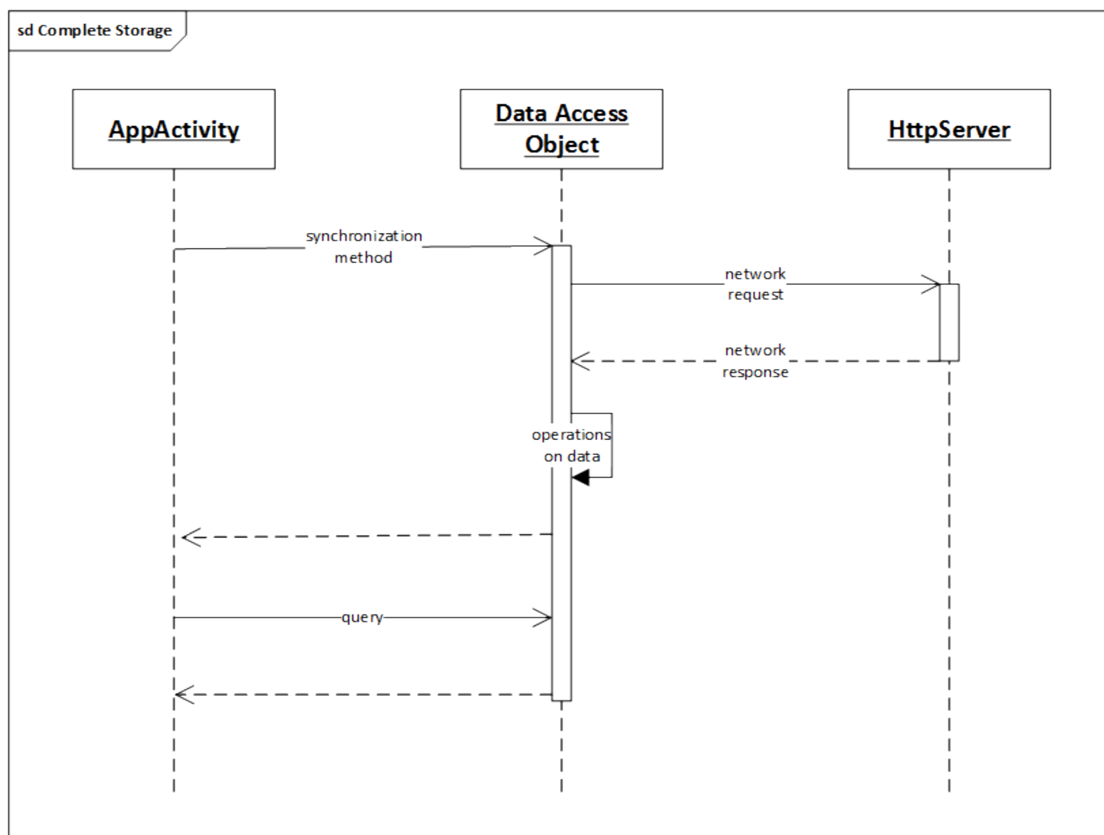


Figure 4.4: UML Sequence Diagram for Complete Storage

Solution

An application component (*AppActivity* in the diagram in fig. 4.4) communicates directly with Data Access Objects, which are responsible for data synchronization and persistence. The adoption of this pattern enables storing the entire dataset on the devices, incrementally or with a single transfer.

Resulting Context
With the application of Complete Storage, the next state of the application features the requested contents. The problematic task is to determine the freshness of data: in fact, since data may refer to an earlier synchronization event, it can be stale. The adoption of some counter-measures may consist in storing information related to transfers to be visualized with the actual data.

Rationale
As mobile devices are becoming powerful computational tools covering more and more use cases, offline work represents a valuable feature. Complete Storage pattern provides a straightforward solution to make all the data available at any time, regardless of the network availability.

4.3 Patterns By Transfer Logic Sophistication

4.3.1 Full Transfer

Intent
The synchronization process transfers the entire dataset at once between the device and the remote system.

Problem
Sometimes designing a deep reconciliation scheme is a painful and useless task, especially if the dataset is prone to change entirely after a modification triggered by manual or automatic updates. A simple reconciliation mechanism consisting in the dataset overwriting may represent an effective solution, also facing the problems due to dataset corruption.

Context
The context resembles the one described for Complete Storage. To apply Full Transfer the dataset of the application must be small enough that it can be downloaded or uploaded in one transfer.

Forces

- Designing a fine-grained reconciliation scheme takes time and resources.
- A small dataset allows its transfer between the device and the remote system.
- If data is not prone to change, transfers are occasional, so their size may not represent a big concern.

Visual Explanation

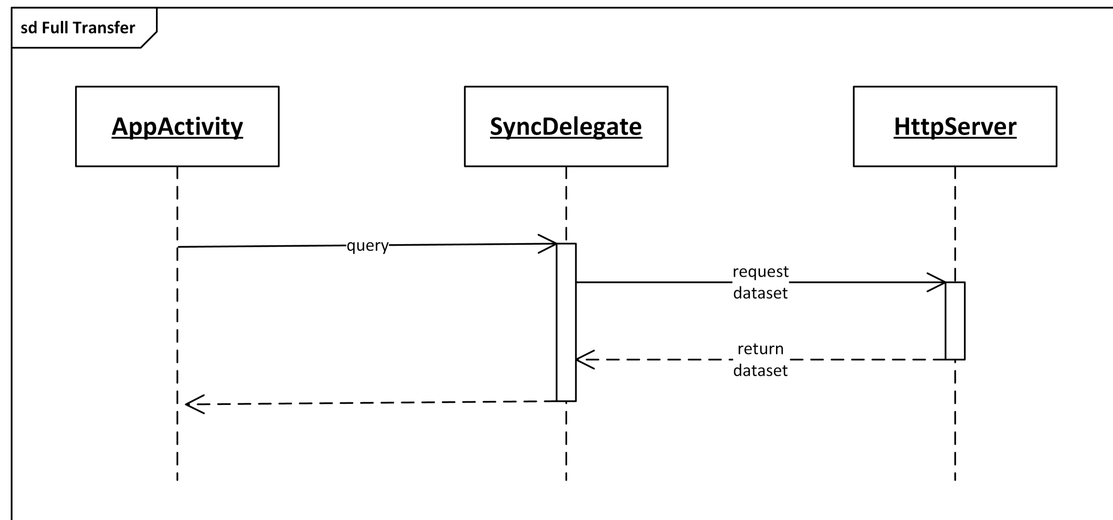


Figure 4.5: UML Sequence Diagram for Full Transfer

Solution

An application component (*AppActivity* in the diagram in fig. 4.5) communicates directly with a generic object handling the synchronization event (*SyncDelegate* in the diagram) in such a way to transfer the entire dataset in a single transfer, with no further granularity.

Note that the visual representation keeps the scenario as abstract as possible, but in place of the *SyncDelegate*, depending on the implementation, we could have an object enabling synchronous or asynchronous synchronization. In this description the time management of the transfer is out of scope though, as the focus is on the transfer logic.

Resulting Context

With network connectivity available, fresh data is provided. Due to the simple nature of the reconciliation scheme, the transfer may require some time to be completed, depending on the dataset's size. Data being transferred may be redundant, if only partially modified since the last synchronization event: in the resulting context this does not change anything dramatically in terms of visualized contents, but bandwidth is wasted and, depending on the synchronous or asynchronous implementation, the UX is altered.

Rationale
The need for data synchronization does not always translate for the need of a complex, fine-grained reconciliation scheme. Full Transfer provides a simple solution, still able to cover realistic scenarios in which the dataset size is limited.

4.3.2 Timestamp Transfer

Intent
For each time a synchronization event is triggered, only the parts of the dataset changed since the last synchronization are transferred between the mobile device and the remote system, taking advantage of a *last-changed* timestamp.

Problem
Considering the issues of network availabilty, speed and bandwidth affecting devices used in mobility, the amount of data transferred to reconcile datasets between a device and a remote system should be minimized. The aforementioned Full Transfer pattern wastes too many resources and the put some notable constraints over the dataset size and dynamicity. Since it is still imperative to synchronize data, another method is needed to reduce data transfer.

Context
The typical context suggesting the application of Timestamp Transfer features contents that are prone to change frequently over time, with remarkable granularity. Long lists of entries that are individually editable, or feeds whose use cases require frequent or periodic data refresh.

Forces

- Information streams and Instant Messaging applications require fine-grained mechanisms to drastically reduce redundancy in data transfers.
- Large datasets are quite common, and applying Full Transfer is neither feasible nor convenient.
- If data is prone to change frequently over time, many transfers are expected to happen, so their size should be minimized to preserve responsiveness.

Visual Explanation

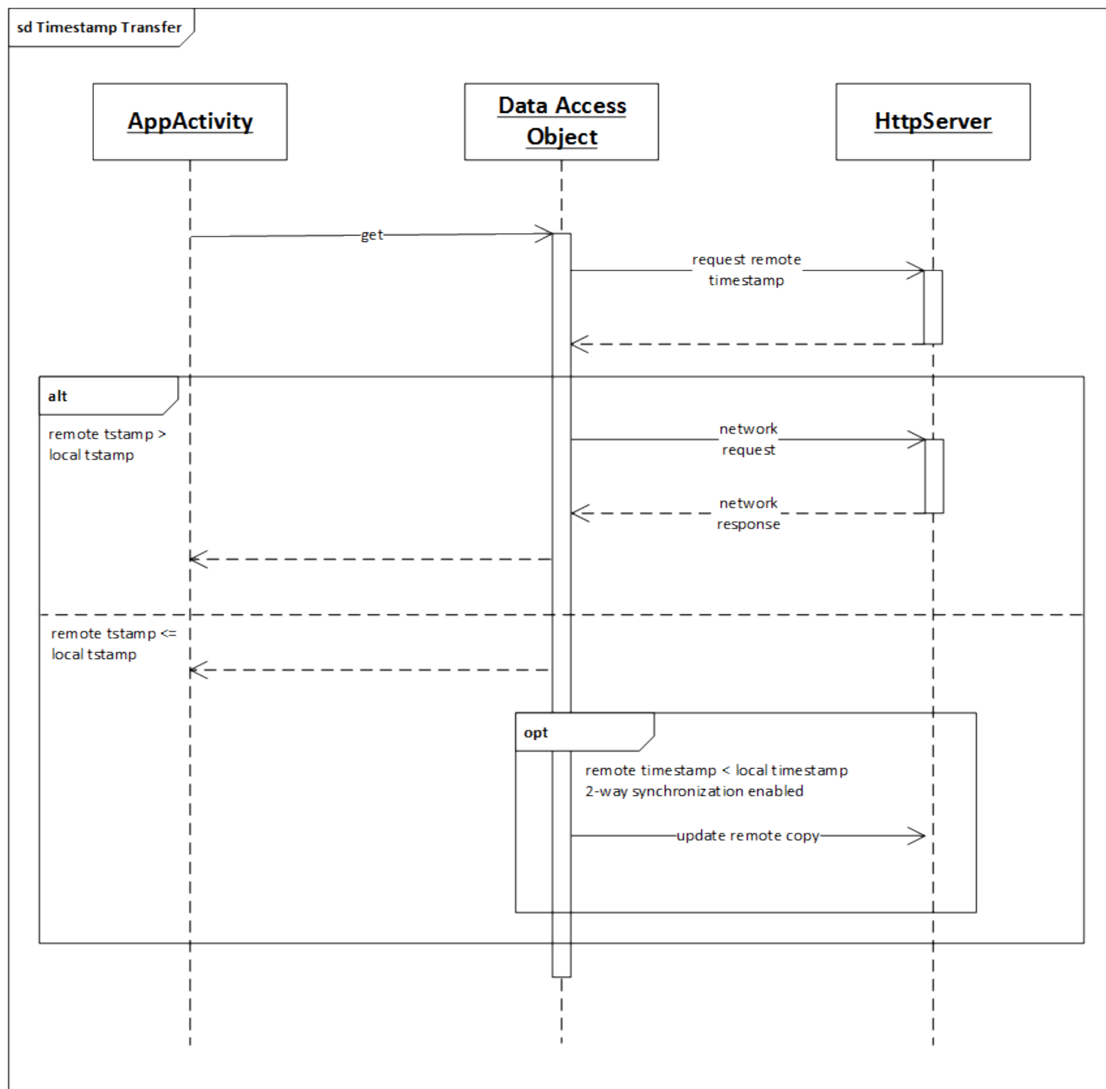


Figure 4.6: UML Sequence Diagram for Timestamp Transfer

Solution

The remote system provides a timestamp to the Data Access Object, that performs the comparisons with the local one. In case of more recent data stored on the remote system, synchronization is triggered. The remote system returns only data that has been added or changed after the local timestamp. In the case of two-way

synchronization enabled, the device submits data that has been created or updated since the last successful submission.

Resulting Context
Generally, having applied Timestamp Transfer, the next state of the application is expected to include fresh data appended to less recent content.

Rationale
Timestamp Transfer pattern approaches to transfer logic resembling the approach of Asynchronous Synchronization to time management, meaning that it aims at optimizing not just the performance, but also non-functional aspects. Thanks to the timestamp exploitation, this differential transfer approach is also extremely flexible, not imposing strictly constraints on the data model.

4.3.3 Mathematical Transfer

Intent
For each time a synchronization event is triggered, only the parts of the dataset changed since the last synchronization are transferred between the mobile device and the remote system, using a mathematical method.

Problem
Having already recognized the issues of network availabilty, speed and bandwidth affecting devices used in mobility, the amount of data transferred to reconcile datasets between a device and a remote system should be minimized. Mathematical Transfer pattern provides an alternative approach for scenarios not complying with timestamps exploitation. For instance, checksums of large pieces of data could be compared just by transmitting the checksum to the remote system, saving a remarkable amount of bandwidth.

Context
The context is close to the one illustrated for Timestamp Transfer, with the difference that the data structure does not allow a discrimination based on history.

Forces

- Transfers' size should be minimized.
- Complex comparisons to determine the differential bundles should be performed at server-side.

- Some type of data (e.g. large binary data) is not eligible for timestamp comparison.

Visual Explanation

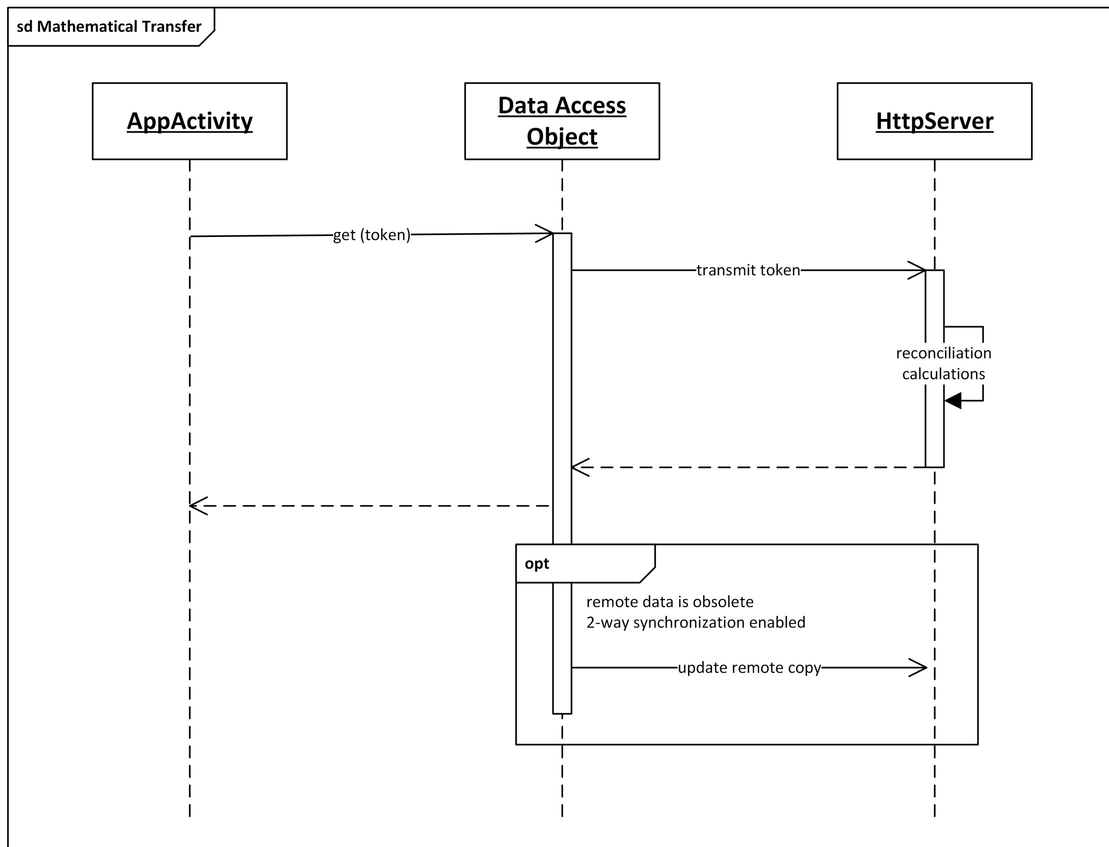


Figure 4.7: UML Sequence Diagram for Mathematical Transfer

Solution

A mathematical method or algorithm is applied to a token passed by the Data Access Object and, based on the outcome computed on the remote system, decides what has to be transferred between the device and the remote system to achieve data alignment.

Resulting Context

Usually, having applied Mathamtical Transfer, the next state of the application is expected to include fresh data, which has been incrementally updated thanks to the exploitation of a mathematical algorithm.

Rationale

Mathematical Transfer can be considered as a generalization of the Timestamp Transfer. Not relying on timestamps, this pattern is even more versatile and may result extremely effective when having to update large binary files. The problematic aspect is given by the scarce re-use potential of mathematical methods, since they typically vary for different types of data. In addition, these methods often require more time to be developed and involve proportionally more steps than Timestamp Transfer's reconciliation processes.

Chapter 5

Patterns Composition

5.1 Building the Big Picture

In our study so far, we have been exploring several aspects of data synchronization in mobile devices. We have been able of distinguish some scenarios, perspectives, phases and scopes of this high-complexity topic.

Our analysis of data alignment mechanisms in commercial applications let us identify some high level behaviors, which inspired us through the illustration of the triggering dimension of synchronization processes. This dimension meets the proper focus under the User Interaction lens, under which we tried to identify and present some simple patterns.

Zooming on the core phase of the process, we have seen how fuzzy the data synchronization logic is, identified its main elements and reviewed some dimensions of client-server communication under another perspective, to better shape our scheme.

Guided by the analysis made on architectural aspects and logic, we deepened our knowledge of the problem of data synchronization, whose study has produced some useful patterns already known in literature. We illustrated them and tried to expand their coverage with some proper visual representations in Unified Modeling Language.

Through this and the next chapters, we would like to realize a complete model of data synchronization in mobile applications. To getting started with this task, we are using the outcomes of what we have experimented with so far as cornerstones, to build something new. The first step will be studying the orthogonality of interaction and synchronization patterns introduced so far. Eventually, we will try to combine known patterns to build remarkable new composite patterns, finding some significant implementations in real-life applications.

5.1.1 Tabular Synthesis

The following table combines the patterns we identified and illustrated in our study so far: as rows, we listed User Interaction Patterns, while as columns we have Data Synchronization Patterns. All cells having a check as value are indicating a possible match in the adoption of the corresponding pair of patterns. Let us state that orthogonality is remarkable, since we register a noteworthy amount of checks for each row/column.

In the next section, using this tabular view as a starting point, we are going to introduce a set of new, complex patterns using User Interactional and Data Synchronization ones as basic building blocks.

Table 5.1: Synthesis of the compatibility between User Interaction and Data Synchronization patterns

	Time		Storage Strategy		Transfer Logic		
	Synchronous	Asynchronous	Partial Storage	Complete Storage	Full Transfer	Timestamp Transfer	Mathematical Transfer
Application launch	✓	✓	✓	✓	✓	✓	✓
Content scrolling		✓	✓	✓		✓	✓
Context change		✓	✓	✓		✓	✓
Pull-to-refresh		✓	✓	✓	✓	✓	✓
Form submission	✓	✓	✓	✓	✓	✓	✓
No interaction (push)		✓	✓	✓	✓	✓	✓
No interaction (pull)		✓	✓	✓	✓	✓	✓

5.2 Composite Patterns Conception

Pattern Form. Patterns are illustrated in compliance with the Coplien Form we already adopted to present User Interaction and Data Synchronization patterns. We slightly alter the pattern form by adding a visual explanation expressed by IFML and UML diagrams, so the resulting structure is arranged as follows:

Problem
Statement of the problem(s) that the pattern works to solve.

Context
Definition and constraints of the specific context of applicability of the pattern.

Forces
Scenarios and motivations supporting the pattern logic or application.

Visual Explanation
Diagrams or graphical representations explaining the pattern and a description of the interaction between the different elements of the pattern.

Solution
Explanation of how the pattern solves the problem considering the constraints of the context.

Resulting Context
Definition and constraints of the context resulting from the application of the pattern.

Rationale
Philosophy inspiring the pattern.

5.2.1 Pattern: Application Launch Synchronous Synchronization

Problem
We want to load fresh data on application launch, so we need to perform synchronization as soon as possible.

Context
It can be a blank container waiting for its composition, or a splashscreen displayed while data alignment is performed.

Forces

- Having up to date content displayed at application launch improves the perception over user experience.
- Occasionally some functionalities are inhibited without fresh data, so – if it is the case – the soon it is obtained, the better the application works.
- Having data aligned at application launch gives the user the opportunity not to refresh manually.
- Synchronize data synchronously in this scenario may be the best option, if stale data is discarded periodically or useless in terms of functionality.

Visual Explanation

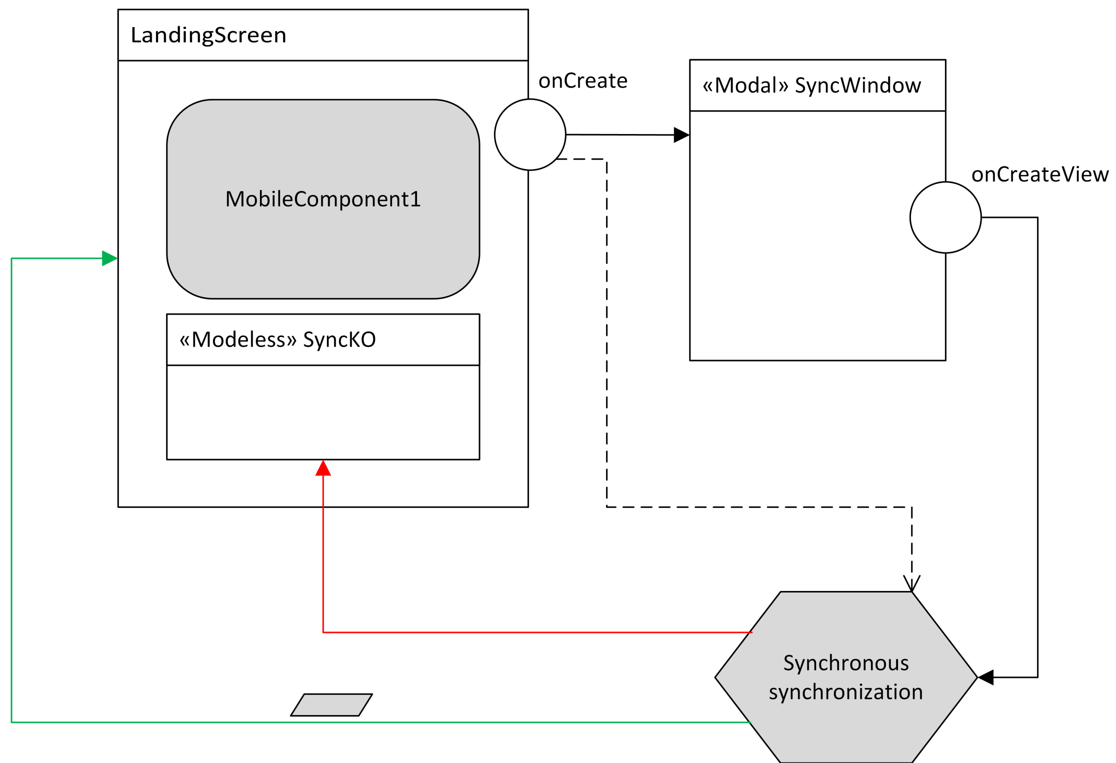


Figure 5.1: IFML diagram of Application Launch Synchronous Synchronization

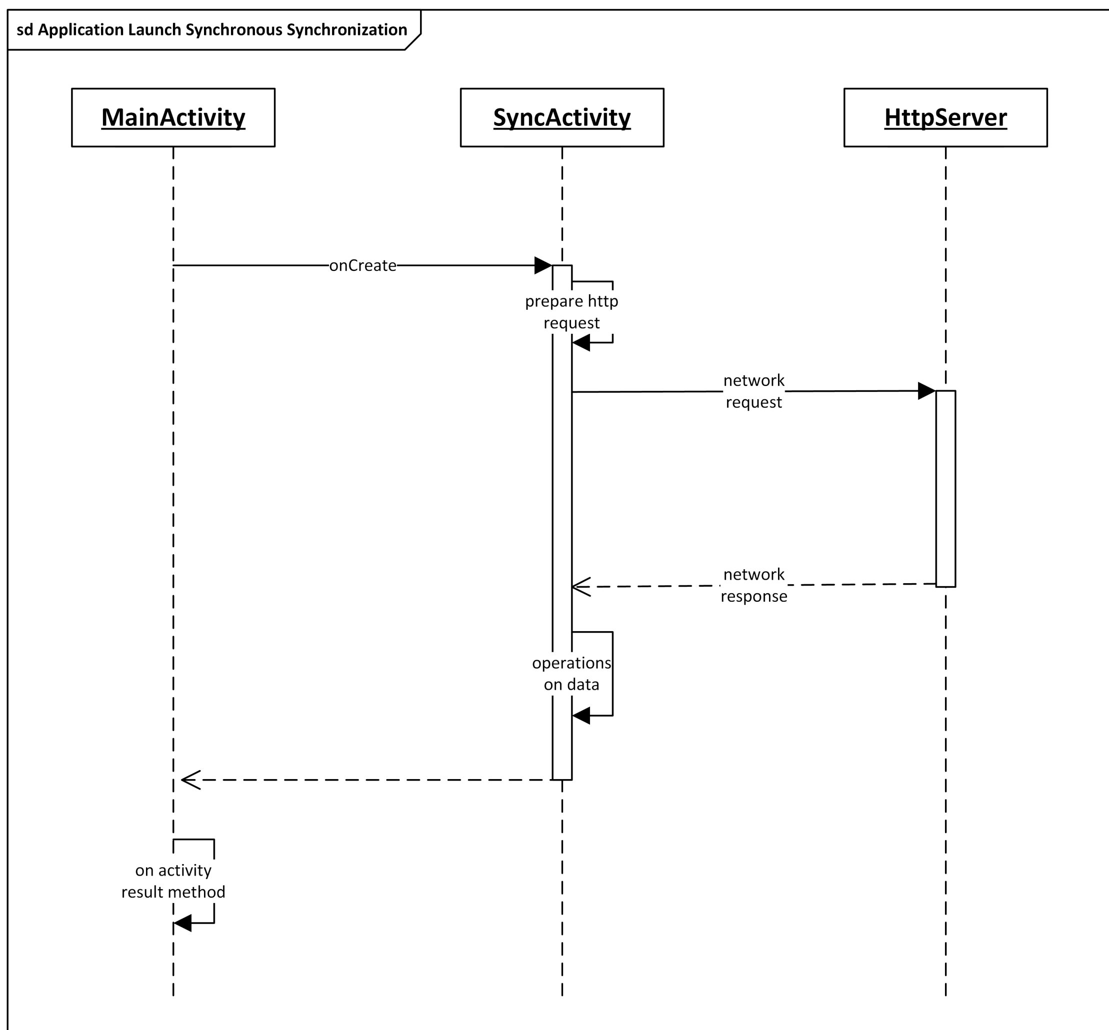


Figure 5.2: UML Sequence Diagram for Application Launch Synchronous Synchronization

Solution

As soon as the composition of the view (the starting point of its lifecycle) starts, the synchronous synchronization event pattern is applied (refer to *onCreate* event in fig. 5.1 on the previous page), with a modal window popping over the active screen and blocking the rest of the interface. As the window's view is composed (*onCreate view* event), the synchronous synchronization event is triggered. Focus on that screen is restored when the synchronization process is concluded, successfully or not. Of course, after a positive outcome, the view composition is completed and contents are displayed (in the UML sequence diagram in fig. 5.2 these operations are generically encapsulated in *on activity result method*). In terms of transfer strategy, this pattern may be coupled with Full Transfer.

Resulting Context
Main activity context of the application, populated with up to date content. Rationale

Rationale
If fresh data is a functional requirement, then waiting for its availability may represent a solution, given that the trade-off between user experience and functionality is considered acceptable.

Asynchronous Variant

Problem
We want data refresh to happen as first task, when launching the application, but without interfere with the rest of the interface, possibly populated with stale or local data, instead of being displayed blank.

Context
The main view of the application, displaying basic containers comprising stale or offline data.

Visual Explanation

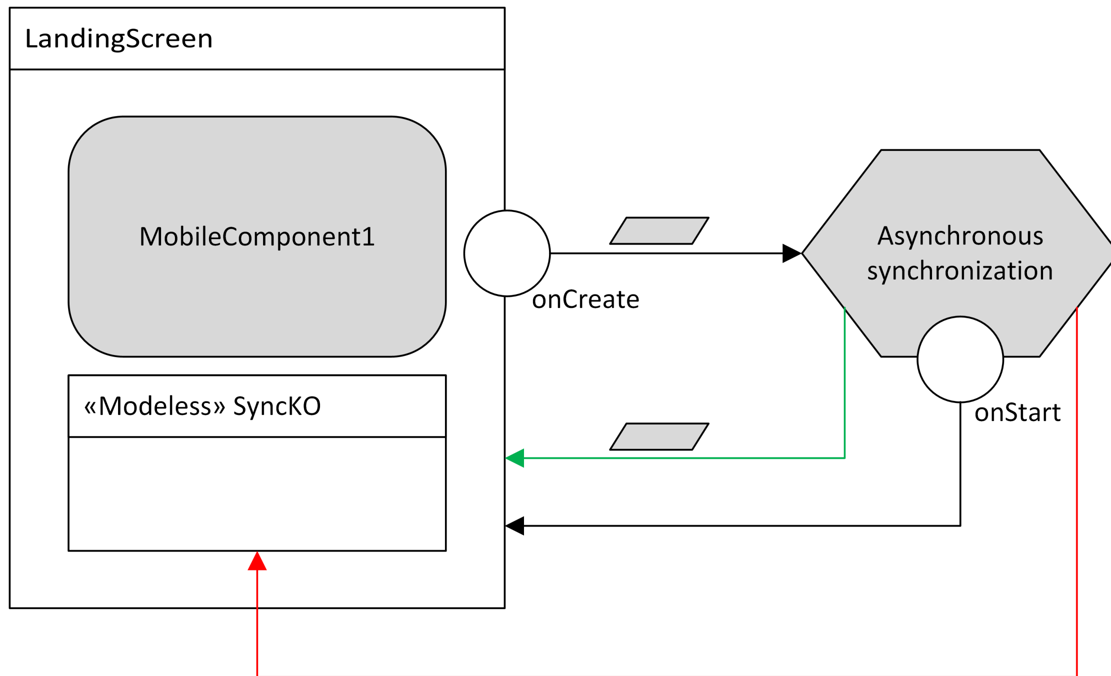


Figure 5.3: IFML diagram of Application Launch Asynchronous Synchronization

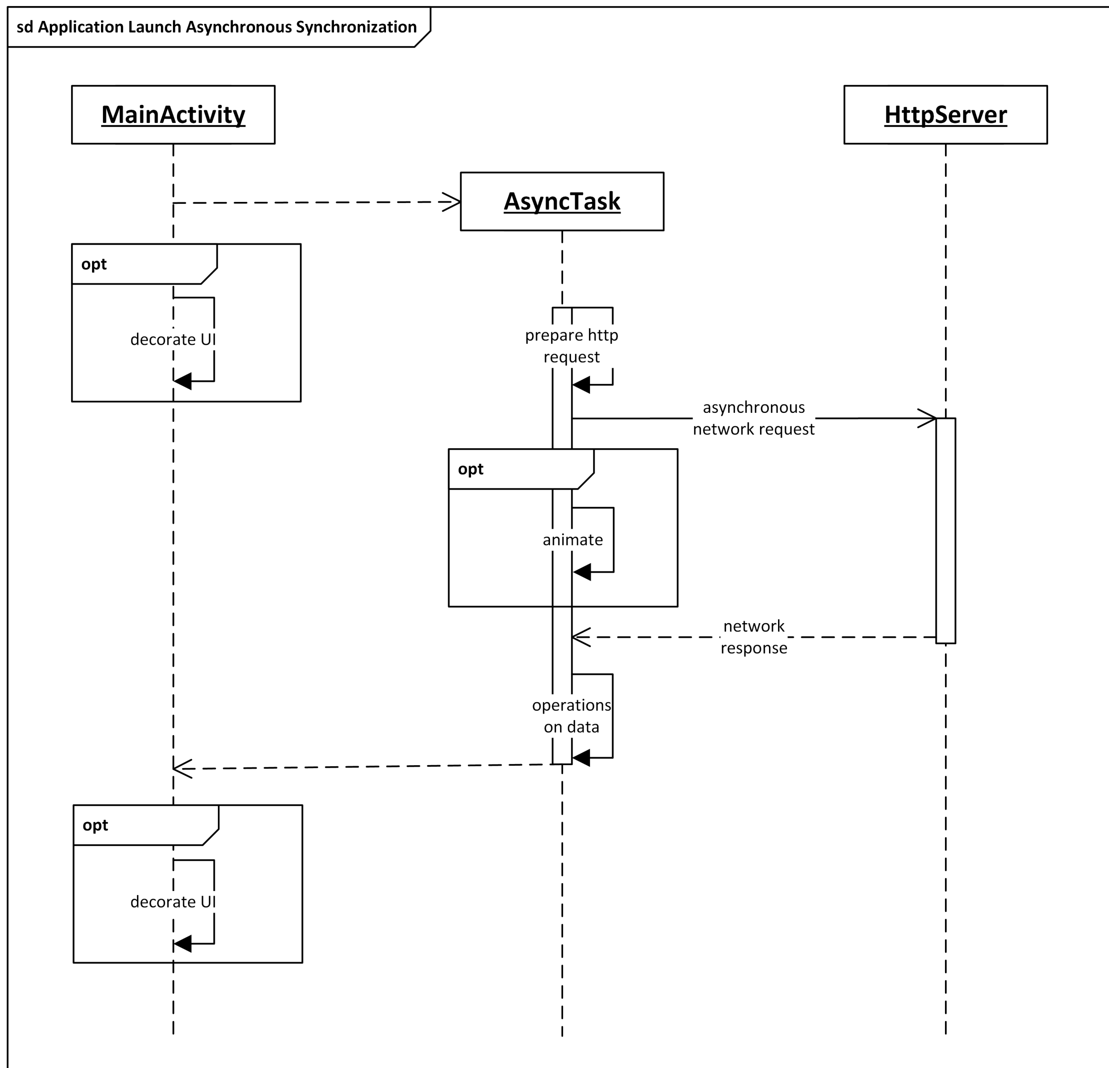


Figure 5.4: UML Sequence Diagram for Application Launch Asynchronous Synchronization

Solution
As soon as the composition of the view (the starting point of its lifecycle) starts (*onCreate* event in fig. 5.3 on page 66), the asynchronous synchronization event pattern is applied, having focus on the main view container restored as the synchronization process starts. During data alignment process, the lifecycle of the application proceeds normally, as emphasized by the use of asynchronous messages visual notation in fig. 5.4 on the preceding page. By looking at the same diagram, we acknowledge that, optionally, the UI may be decorated before, during and after the synchronization process. At synchronization completion, the contents displayed in the view components are refreshed (refer to the success flow in fig. 5.3 on page 66, featuring a *ParamterBinding*), while a negative outcome may be signaled by means of a modeless window implementation (error flow in fig. 5.3 on page 66). Transfer logic (full or differential) may vary in accordance with requirements.

Rationale
Renewing data at application launch is generally a good practice, and realizing this process using an asynchronous implementation is even better if some of the components of the interface are functional, regardless of the obsolescence of data.

5.2.2 Pattern: Content Scrolling Asynchronous Synchronization

Problem
We want to synchronize elements presented in a scrollable view gradually, without interrupting the fruition of contents (lazy policy).

Context
Browsing a list – or, more generically, a scrollable container – comprising contents to be synchronized.

Forces

- Synchronize all the content in a single transfer can be inconvenient, in terms of performance, bandwidth and storage consumption.
- Requesting ad-hoc interaction to perform data synchronization or refresh breaks the user experience.
- Data synchronization should remain transparent with respect to the user.

Visual Explanation

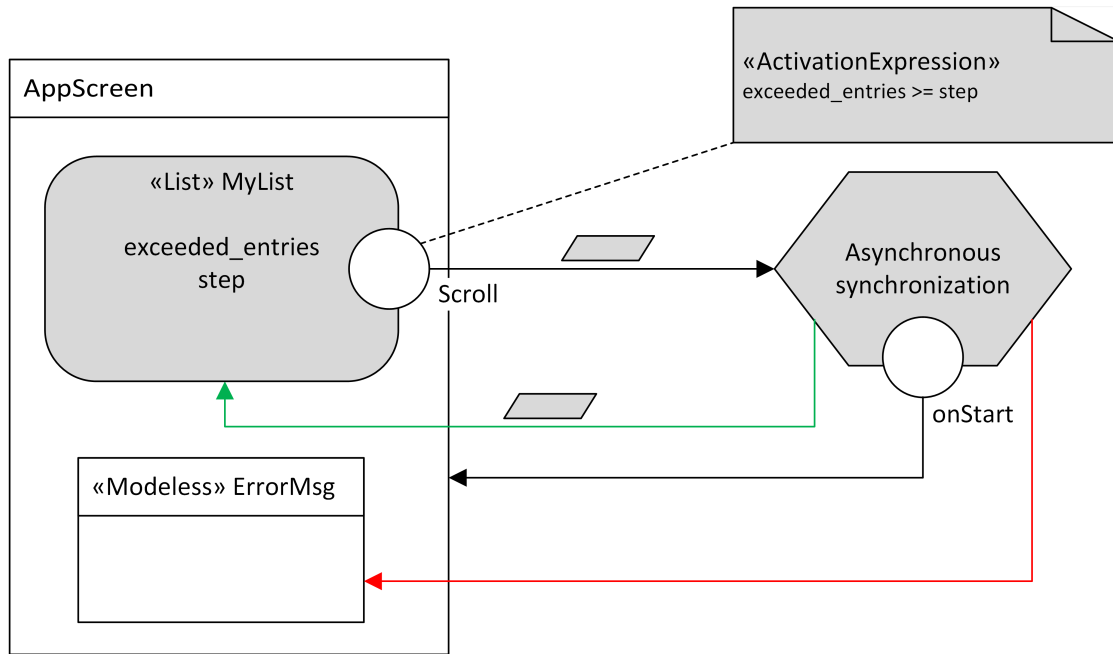


Figure 5.5: IFML diagram of Content Scrolling Asynchronous Synchronization

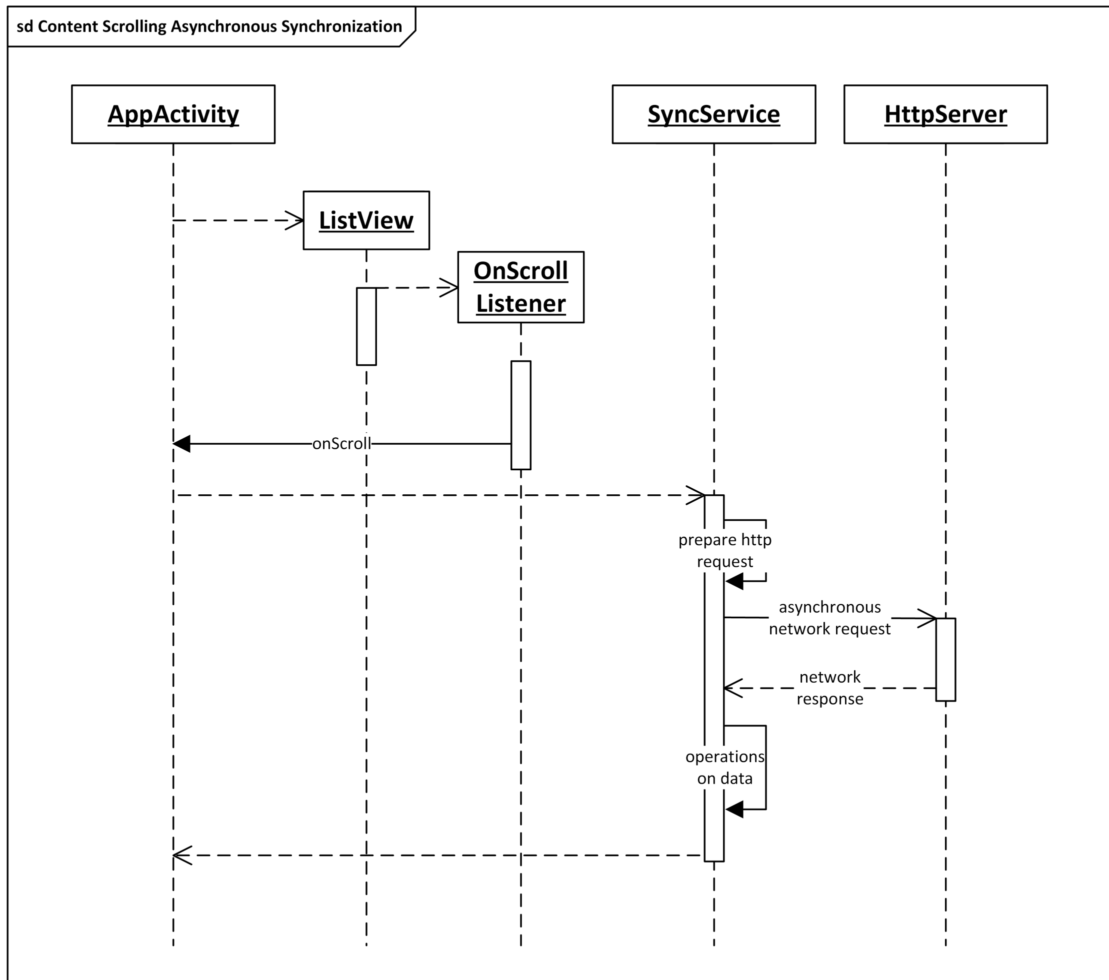


Figure 5.6: UML Sequence Diagram for Content Scrolling Asynchronous Synchronization

Solution

Data Synchronization starts in response to the scroll event, given that the activation expression is satisfied. We are not considering the direction of the gesture, since the pattern applies to both horizontal and vertical lists. Practically, when the user exceeds the minimum amount of entries (attribute *step* in fig. 5.5 on the previous page) requested to trigger the event, the asynchronous synchronization starts. The evaluation of this condition is performed by an object that we named *OnScrollListener* in fig. 5.6. We want to underline that the focus stays on the view container the user is browsing (i.e. *AppScreen* in fig. 5.5 on the previous page), as indicated by the navigation flow triggered by the *onStart* event of the synchronization action. When the action is completed, a positive outcome results in passing new data to *MyList* view component, while a negative one fires the

visualization of a modeless windows (e.g. a "toast notification" in Android) showing an error message. The transfer is differential and it can encapsulate look-ahead policies.

Resulting Context
The context remains unaltered, as the user would expect it. New content is loaded and displayed.

Rationale
Loading content is a responsibility of the application components: users should not be requested to perform ad-hoc actions, so this pattern exploit the natural interaction as a trigger. To achieve all the tasks requested by synchronization process seamlessly, without blocking the interface, all the operations are executed in asynchronous way. The only downside is given by latency times, possibly postponing the invalidation or the incremental population of the list.

5.2.3 Pattern: Context Change Asynchronous Synchronization

Problem
Modifying the context of exploration in application relying on synchronized data visualization and manipulation often leads to the need for a data alignment event. This process must be the most seamless and transparent possible, to preserve the smoothness and continuity of the user experience.

Context
The context resembles a hypertext: interacting with elements in the container mimics the behavior of navigation links in a web page.

Forces

- Differential transfer is choice when the application does not need to display all of its contents in a single view.
- Exploiting screen transitions has two advantages: eliminating the need for a dedicated interactor and coping with the nature of asynchronous synchronization event.

Visual Explanation

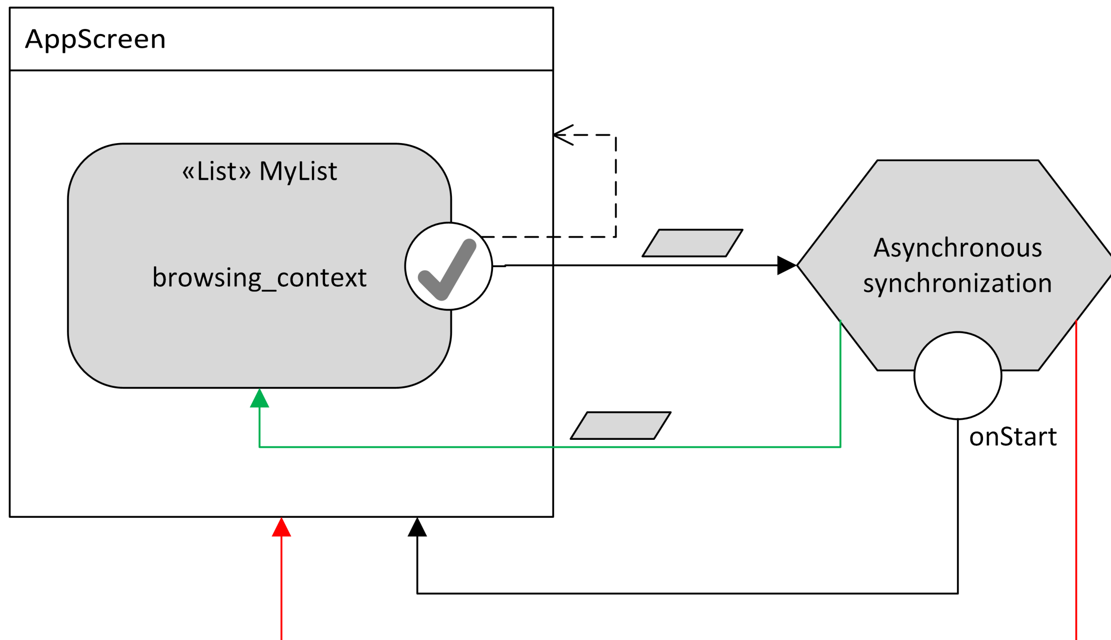


Figure 5.7: IFML diagram of Context Change Asynchronous Synchronization

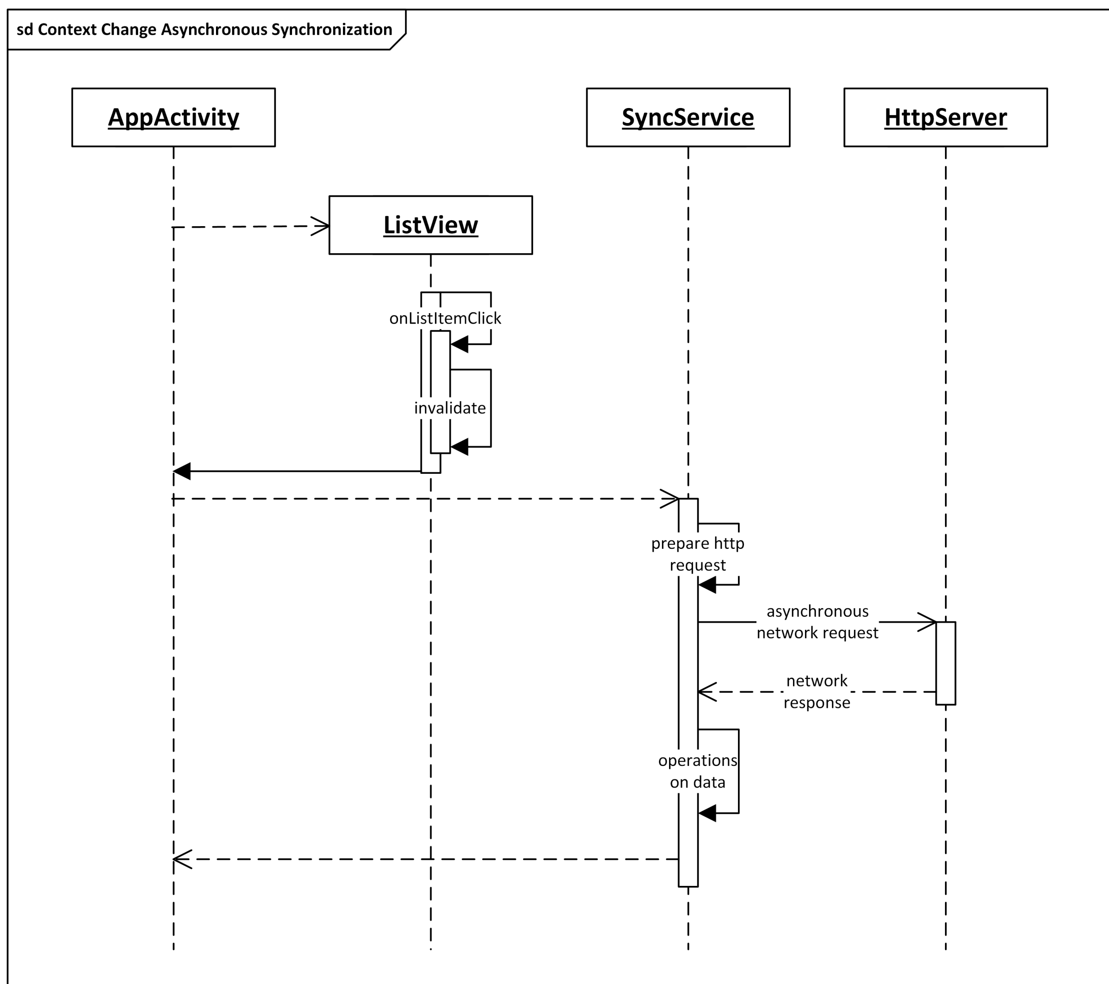


Figure 5.8: UML Sequence Diagram for Context Change Asynchronous Synchronization

Solution
Selecting an entry in the list (which behaves as a link in a hypertext) triggers the context change (involving some parameters passing to the container comprising the list view component, as depicted by the data flow connecting the selection event to *AppScreen* in fig. 5.7 on the facing page) and the asynchronous synchronization action. In the UML sequence diagram in fig. 5.8, the object named *ListView* handles its invalidation (refresh of its content) as it detects the interaction with one of its entries (*onListItemClick* method call). As the action starts (refer to *onStart* in fig. 5.7 on the facing page), focus is sent back to the view container, not to interrupt interaction. A positive outcome of the completed synchronization process triggers the refresh of the list view component, which loads the new data. The error outcome implementation may vary. The transfer is differential and it

can encapsulate look-ahead policies.

Resulting Context
The frame does not change and, ideally, the interaction pattern remains unaltered and applied to the new content.

Rationale
Client applications adopting the "explorer" paradigm often have to deal with large amounts of data. Synchronizing the minimum amount of needed data at each step, in compliance with the asynchronous synchronization pattern, offers a good solution in terms of responsiveness, bandwidth and storage constraints.

5.2.4 Pattern: Pull-To-Refresh Asynchronous Synchronization

Problem
On-demand synchronization has to be featured as an option, but classical implementations like buttons in menu-bars look outdated and disrupt the user experience. The process must be completely unobtrusive, with minimal feedbacks and no interruptions in terms of functionalities.

Context
Usually a scrollable view container, but it can be a generic one.

Forces

- This pattern is becoming a standard for synchronized lists (e.g. timelines) and, more in general, scrollable components.
- A vertical gesture requires less effort compared to the selection of a traditional entry in a menu, given the morphology of most of mobile devices: as a result, the user experience is more continuous.
- This pattern can replace the "refresh button" in user interfaces, helping to keep them minimal.
- This pattern provides an "infinite" visual feedback signaling the ongoing synchronization process, started at alignment initiation and dismissed at termination.

Visual Explanation

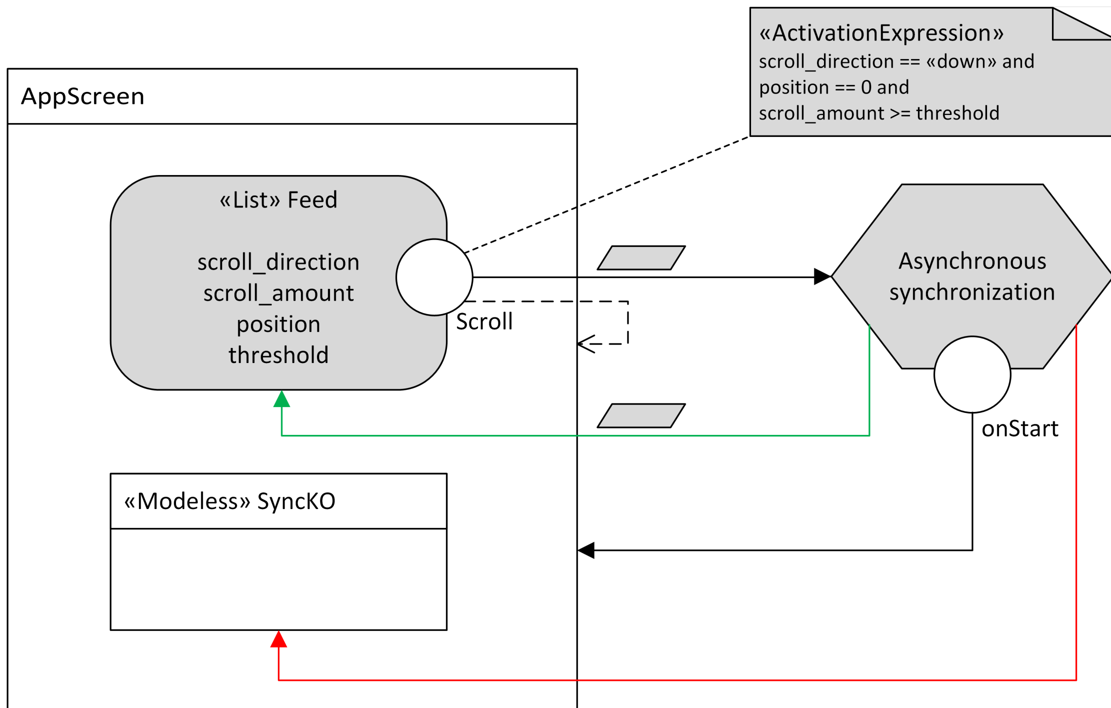


Figure 5.9: IFML diagram of Pull-To-Refresh Asynchronous Synchronization

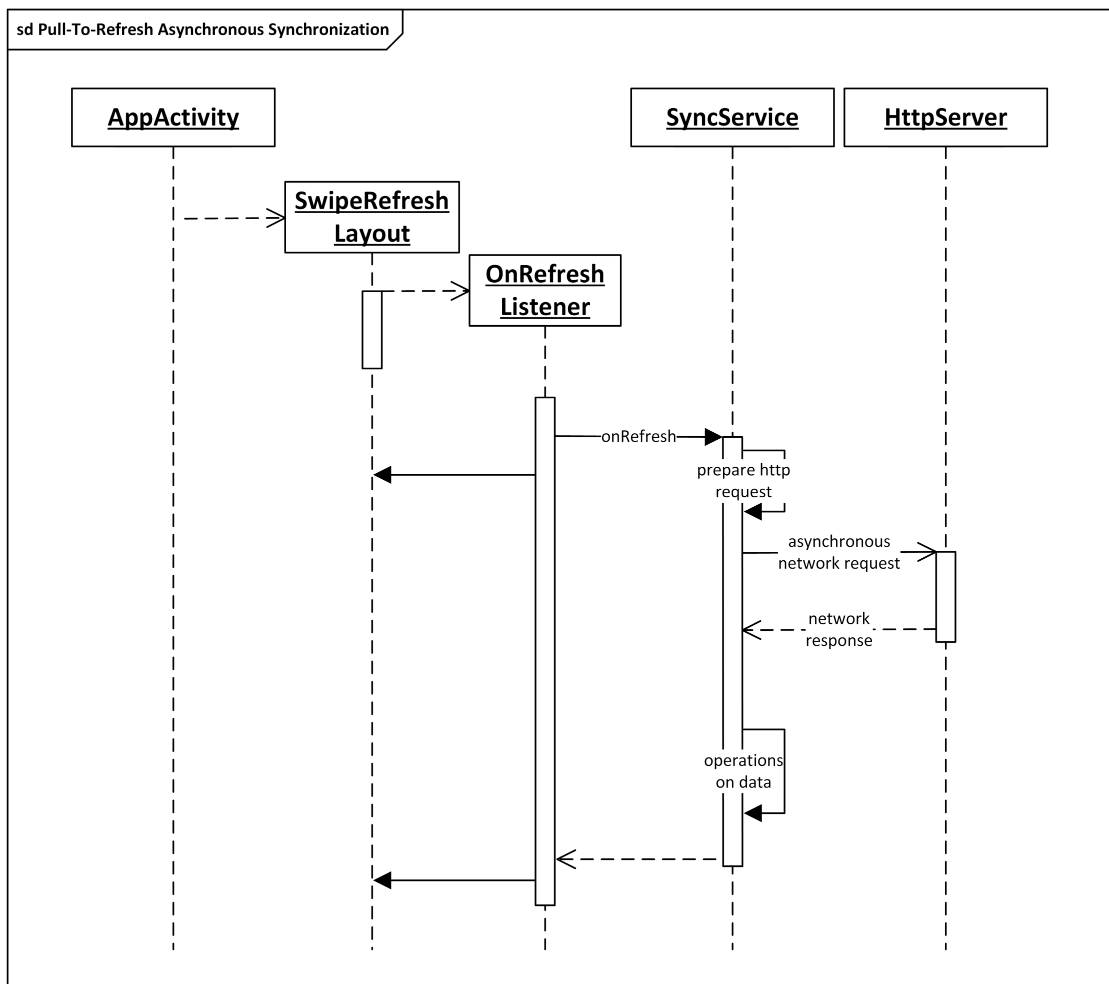


Figure 5.10: UML Sequence Diagram for Pull-To-Refresh Asynchronous Synchronization

Solution

The functionality is intuitive: the asynchronous synchronization process is triggered by the completion of the gestural combination. Specifically, given that the user is pointing at the first element of the list, a fling down gesture makes the list "draggable" from the top to the bottom; to complete the event, the list must be released after having been dragged for a certain vertical distance, greater than the value stored in *threshold* attribute in fig. 5.9 on the previous page). At this point, the listener (*OnRefreshListener* object in fig. 5.10) communicates to its parent to start the animation and asynchronous synchronization pattern is appended. As usual, starting the action restores the focus on the view container, while at completion time the contents are refreshed. A modeless window implementation is displayed in case of errors.

Resulting Context
The pattern turns the target view component into an interactor to trigger the asynchronous synchronization process.

Rationale
This pattern combines the creation of a gestural interactor and the implementation of asynchronous implementation. Thanks to the continuous visual feedback provided, it makes the process duration visible to the user, which is consequently conscious of the process running state.

5.2.5 Pattern: Form Submission Synchronous Synchronization

Problem
We want to validate and refresh data on a server immediately after the form is submitted, possibly signaling the process with a feedback.

Context
The view comprises a form and a submit interactor, whose layout depends on the implementation variant.

Forces

- When user input is required, the most serviceable way to register it is to commit it after manual confirmation.
- A synchronous transfer can reduce the time required by the operation.

Visual Explanation

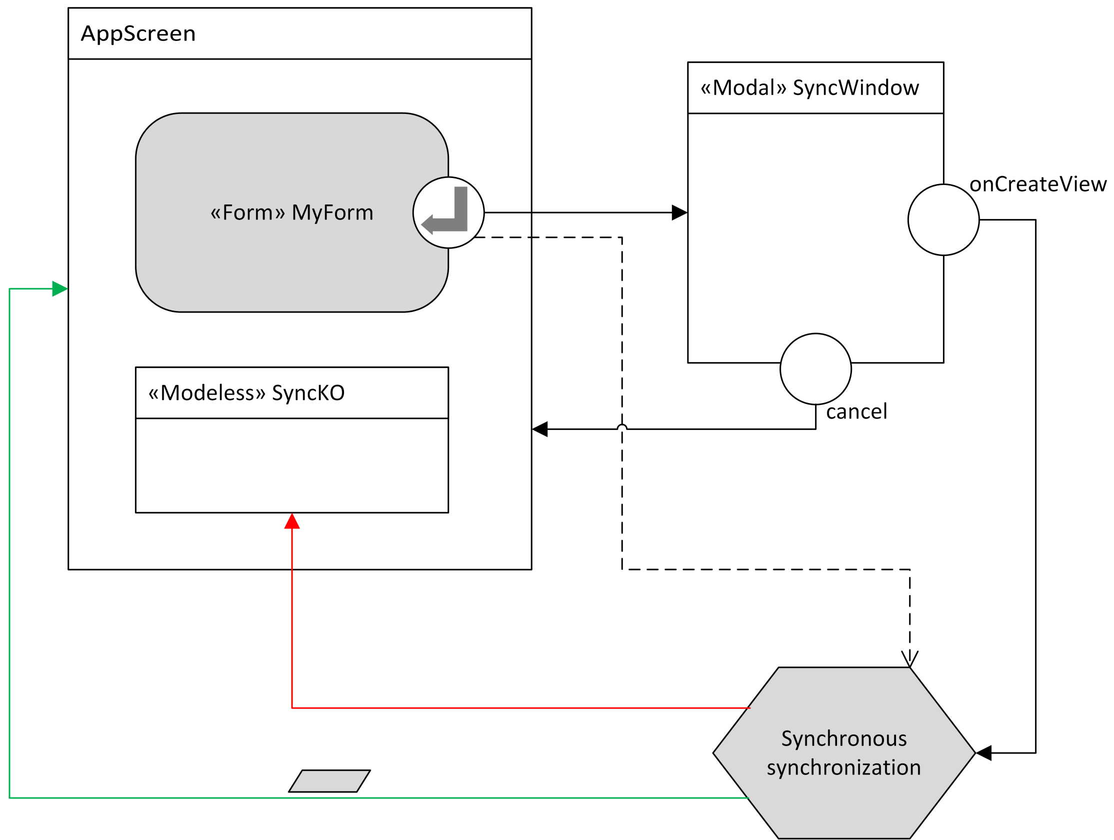


Figure 5.11: IFML diagram of Form Submission Synchronous Synchronization

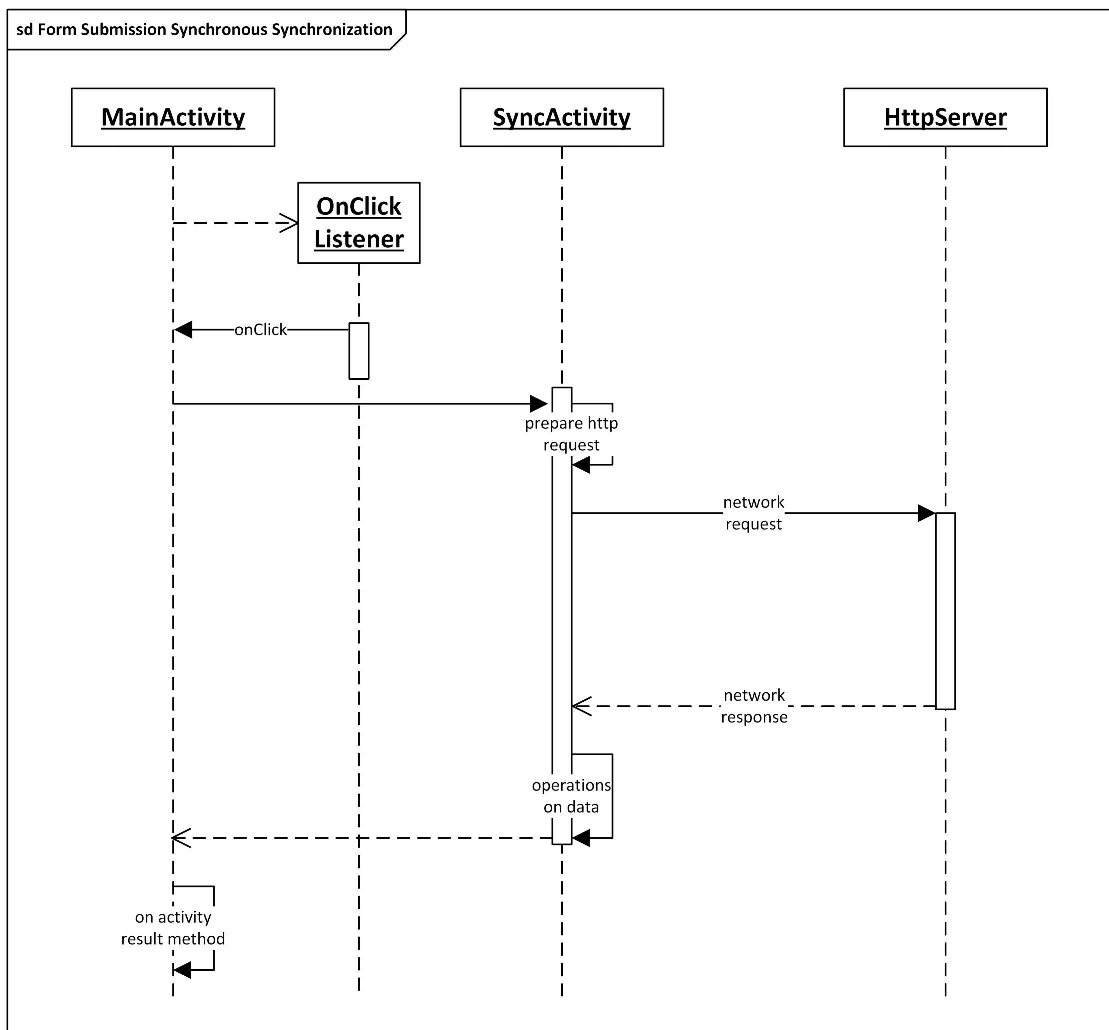


Figure 5.12: UML Sequence Diagram for Form Submission Synchronous Synchronization

Solution

Using a simple, intuitive interactor (the submit event in fig. 5.11 on the preceding page, corresponding to the *OnClickListener* object in fig. 5.12), the data alignment process is manually triggered following the form of the synchronous synchronization pattern, with a foreground modal window interrupting the interaction until process' completion.

Resulting Context

The resulting context may vary according to the implementation logic. For instance, it can comprise a modal pop-up or a visual element pinned to the parent view signaling the operations in progress. However, during the synchronization process,

the user interface is blocked, giving only the possibility to cancel the operation.

Rationale
Committing an operation requires the complete consciousness of the user, whose first concern is typically about performing a wrong action on data. The easiest way to letting her/him know of the synchronization process is indeed the application of this pattern. A synchronous implementation is the best option to monitor and control the process in a simple way.

5.2.6 Pattern: Push-Triggered Synchronization

Problem
We need to synchronize data as the server sends a message to the mobile application, acting as a client.

Context
Generic: the push event may happen at any time, given that network connectivity is available.

Forces

- Push notifications are choice when instantaneous synchronization is a requirement.
- Push notifications are able to trigger the decoration of the application's user interface.
- The user is accustomed to interact with push notifications.
- Push events enable the most advanced form of automated data synchronization, a sort of on-server-demand process.

Visual Explanation

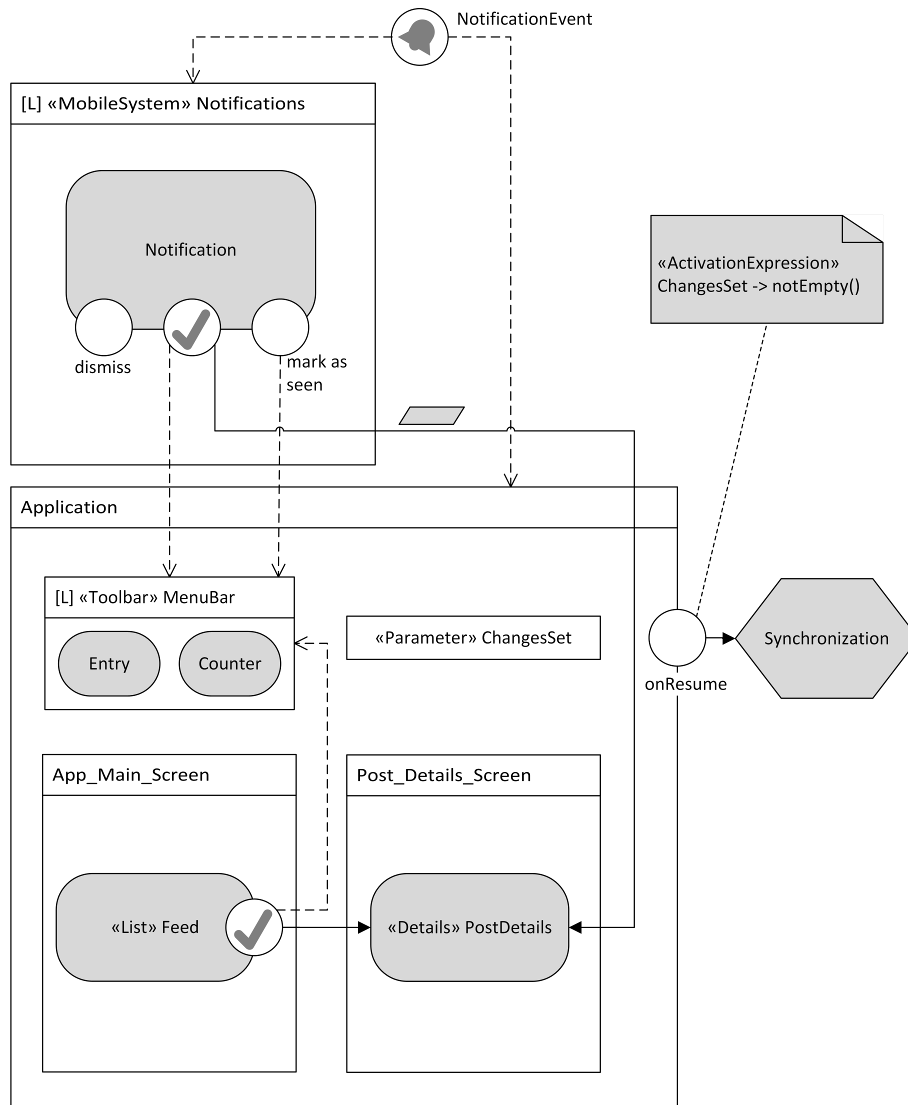


Figure 5.13: IFML diagram of Push-Triggered Synchronization

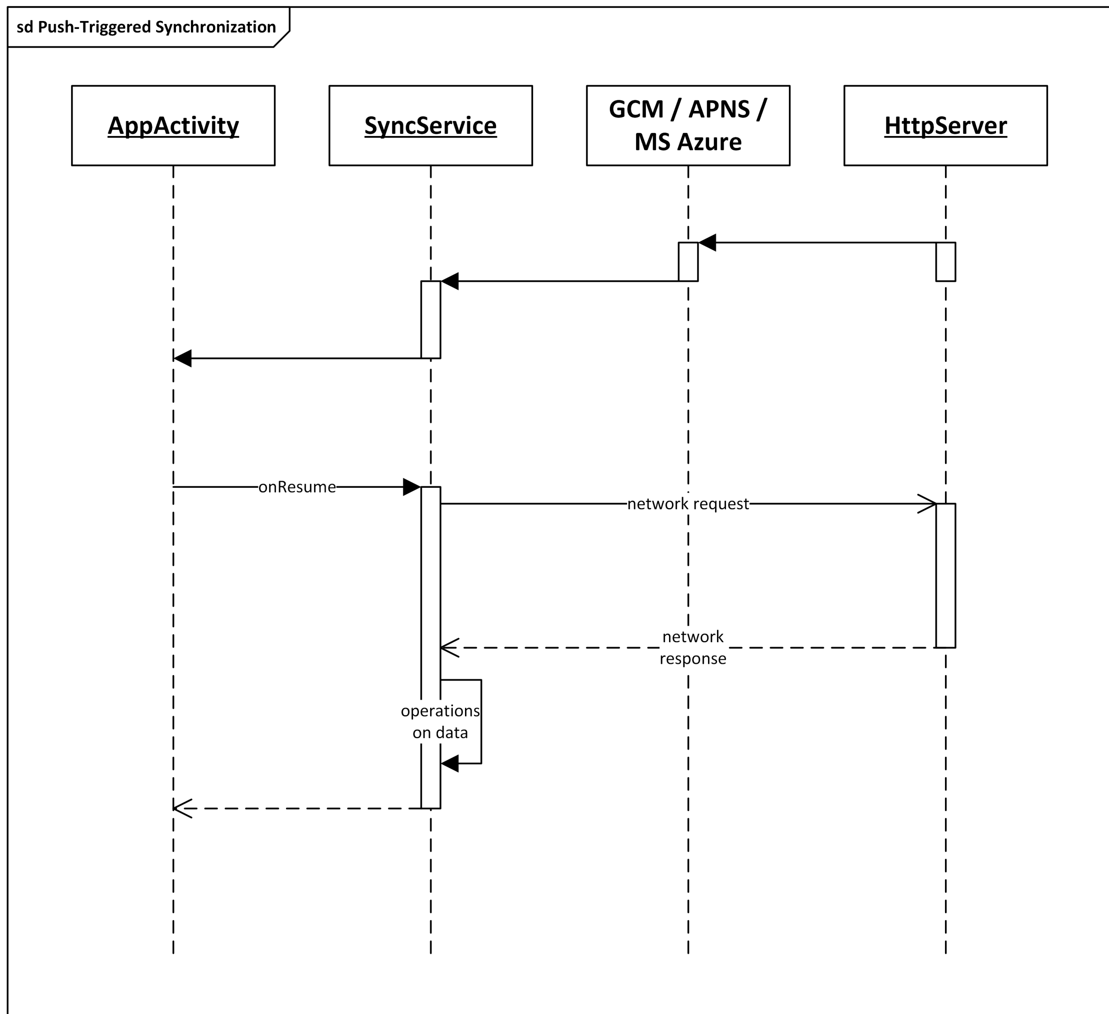


Figure 5.14: UML Sequence Diagram for Push-Triggered Synchronization

Solution

As the push event is caught by the OS, a notification is created within the dedicated notifications container. Depending on the OS, several options can be provided as interactors, but usually a simple touch let the user navigate to the *Details*-stereotyped view component (fig. 5.13 on the previous page), binding some parameters to correctly display information. It is interesting to underline the effects that the push event can have on the application's user interface, which is possibly decorated within its menu bar implementation. In terms of business logic, the push event alters some parameters within the application context. In the diagram above, we stereotyped this event by introducing the collection parameter named *ChangesSet*. Usually, when the application is resumed, the control logic mechanisms verify the

value of the parameter (in this case, its non-emptiness) and, according to the comparison result, determine whether synchronization should be performed. Note that we are not specifying the data synchronization pattern, because in this case, it would be out of our scope. In some cases, for instance having the application running in foreground, the synchronization event is triggered directly by the *NotificationEvent*. The UML diagram in fig. 5.14 on the facing page exposes the server-to-client communication schema, with a mediator interface (Google Cloud Messaging, Apple Push Notification Service or Microsoft Azure depending on the platform) forwarding messages to the receiving device.

Resulting Context

The OS landmarks in the user interface and the application's interface are decorated, and as the application is resumed, its content is refreshed because of data alignment.

Rationale

Push synchronization permits the ideal implementations of the inverse asynchronous synchronization technique, with the server notifying the client of changes on data and triggering the alignment. On the other hand, its implementation introduce some complexity, since request the creation of some dedicated components of the application acting as receivers for push events.

Chapter 6

Application of the Patterns in Mobile Front-end Modeling

The closing chapter of this study features a demonstrative nature, aiming at verifying the applicability of the patterns identified and proposed so far. The research path undertaken for this examination naturally leads to a concrete experimentation within the frame of Model Driven Development for Mobile Applications, maintaining the focus on the Front-end design.

6.1 Software Requirements

The application whose Front-end will be modeled using IFML covers a real-world scenario, being a concept application thought to showcase a popular exhibition having fashion as main theme. The aforementioned event, named "Il Nuovo [vo-ca-bo-là-rio] Della Moda Italiana", will be actually hosted by the "Triennale di Milano" from November 2015 to March 2016.

In this section, we briefly report some of the requirements of this concept application, in order to discuss them and build a conceptual model based on the constraints they dictate:

Informative landing view
This section must include a thematic map of the exhibition and other informative material, comprising an "about" component.

Catalog views for Topics, Exhibitors, Events and Artists
Each room of the exhibition is associated with a single topic. Topics (less than 20) must be listed in a catalog view. Selecting one, the details of the associated room are displayed. The same design must be applied to

exhibitors (100-150, each one is related to one or more topics), whose detail view comprises a map showing the rooms in which it is distributed. A list of the proposed events has to be present, with the possibility to show the details of the event in a view which contains also a form to save the event in agenda. Creation of new events or changes to existing ones should be notified via push notifications. Following the same schema, artists (around 300) must be listed in a dedicated view, featuring the possibility to view their individual page including biography, contact information and a high-quality picture of their artwork.

Integrated dynamic agenda view
This user-centered page is designed to list saved events.

Augmented reality
The application must include an integrated QR-code scanner through which it is possible to decode the labels which will be placed next to each artwork. A successful scan triggers the navigation to the associated exhibitor detail page.

Beacons interaction
The application must react when receiving signals from the sensors that are distributed alongside the exhibition path. Functionally, when a sensor detects a device in its proximity, given that positioning systems are active on the device, a push notification is sent to redirect the user to the page of the associated room.

Social stream and News views
Dedicated views must be provided for a social stream, showing interactions related to the exhibition in real-time, and news. The social stream must be reachable from each exhibitor's detail page: this interaction flow should trigger a mechanism to show social posts related to the given exhibitor. News must be transmitted via push notifications.

For the sake of simplicity, we are assuming that the whole applications' views can be consulted after user authentication, which will not be covered by the model.

6.2 Model Realization

This section portraits an effective demonstration of the application of the patterns we introduced in the previous chapter, emphasizing their introduction in the application model. We take as a starting point the model we realized without integrating data synchronization logic, with the intent of introducing, step by step, adequate patterns to comply with the requirements.

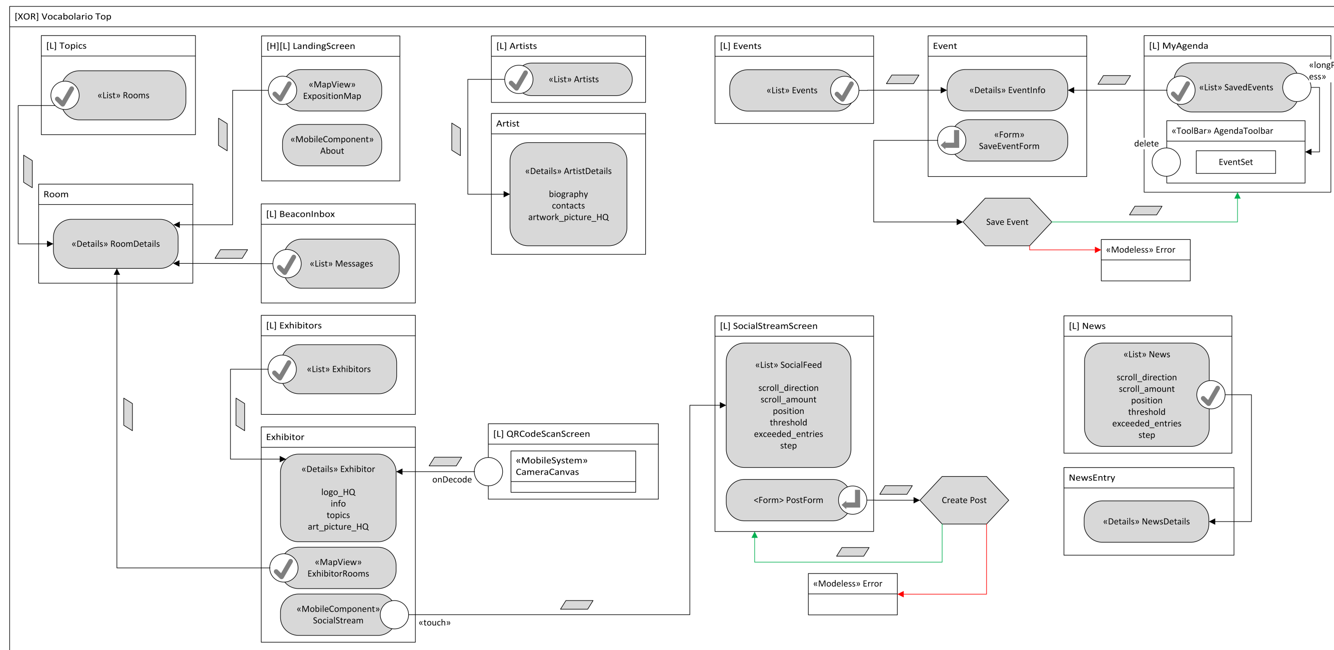


Figure 6.1: Application Front-end model in IFML

6.2.1 Design of Data Synchronization Logic

The application of patterns must strictly match the requirements we have to satisfy in terms of synchronization mechanisms. Few of them are explicitly mentioned within the generic software requirements listed in the previous section, but generally some considerations are mandatory preconditions to determine what are the best options to rely on.
Let us begin by discerning the processes based on their nature.

Instantaneous synchronization. As stated in the requirements, push notifications (and subsequent synchronization events) are a mandatory feature to apply to Events, News and Beacon messages. In this case, we may want to model the synchronization action as a direct response to the notification, simplifying the pattern.

Real-Time synchronization is required for the social stream view, but unfortunately there is no effective way to represent it, since it does not involve interaction at all. However, we would opt for another push mechanism or for an optimized pull one, as the "long polling" implementation already analyzed on Dropbox mobile application, in Chapter 1.

On-demand synchronization. Concerning this type of data alignment, given requirements are very flexible. Nonetheless, as already mentioned multiple times, the reasoning on the best solutions to apply to each component of the application must be strict. In fact, the right choice can make the difference in terms of non-functional requirements in general, which are rather relevant in the Mobile environment.

Based on the fact that most of informative data is rather static (it rarely changes over time), synchronizing it asynchronously at application launch makes sense. The portion of data involved in this process is related to the catalogs views (topics, exhibitors, artists and events).
Details views for artists and exhibitors is downloaded asynchronously as requested (i.e. when the view is created). The same process is performed for individual events and the agenda view.
News and social stream, which are long lists of dynamic data must implement some lazy techniques to download data: exploiting their scrolling is a possible solution. Since these components should feature real-time data alignment, we may want to improve the user experience by adding the feature of manual synchronization via "pull-to-refresh" interaction: in fact, the idea of having the chance of refreshing contents manually reinforces the user's perception of empowerment.

6.2.2 Patterns Application

Patterns for instantaneous synchronization

Event and News entry. Having stated that back-end creation and modification of events and news articles are to be notified instantaneously, Push-Triggered Synchronization Pattern is the trivial solution to apply. Actually, in this case the pattern is highly simplified, to show its behavior within the foreground application: as a matching push notification is received, an asynchronous synchronization event is launched. As illustrated in figure 6.2, with a positive outcome of the process the success interaction flow is taken and the *Details*-stereotyped component in the screen is populated, using a token to match the corresponding event or news article.

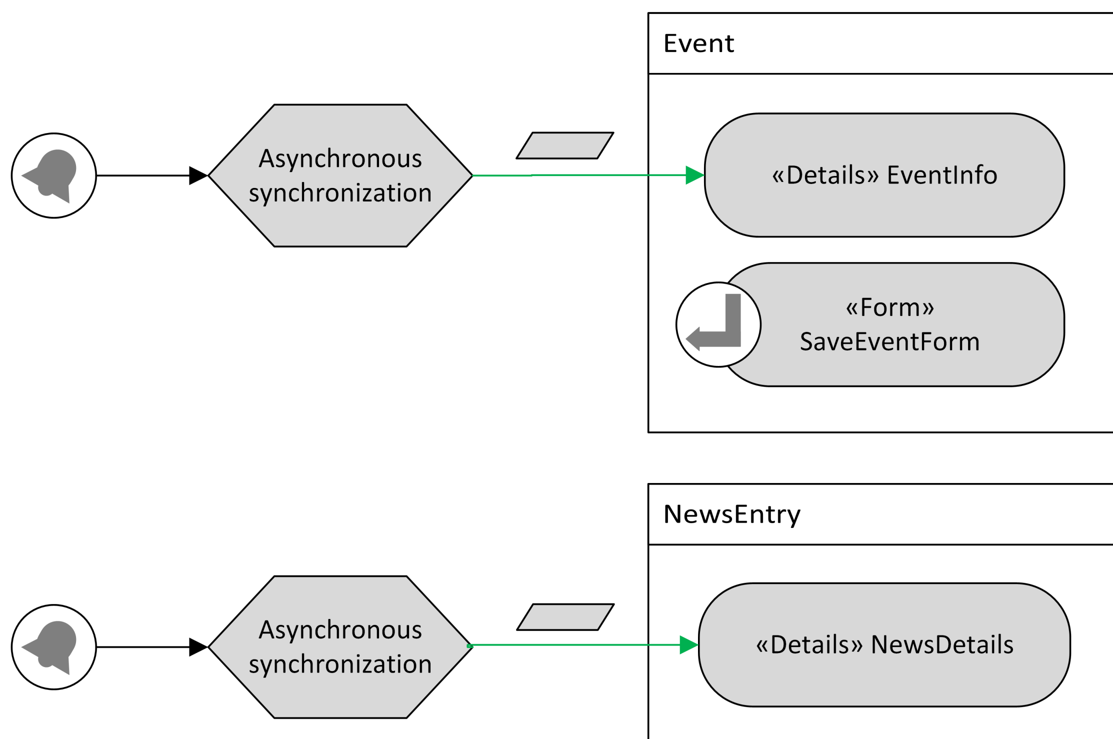


Figure 6.2: Push-Triggered Synchronization variant for event and news entry

Beacons messages. The data alignment event following the reception of a push notification coming from a sensor in the proximity of the device is modeled analogously to the corresponding processes already illustrated for events and news. Specifically, an activation expression acts as process firing condition, stating that synchronization happens only whether the device's positioning technologies are enabled. Furthermore, any event, on successful outcome, appends its message to a list working as an inbox for beacons. In this way, messages remain available for deferred consultation. The model is depicted by figure 6.3.

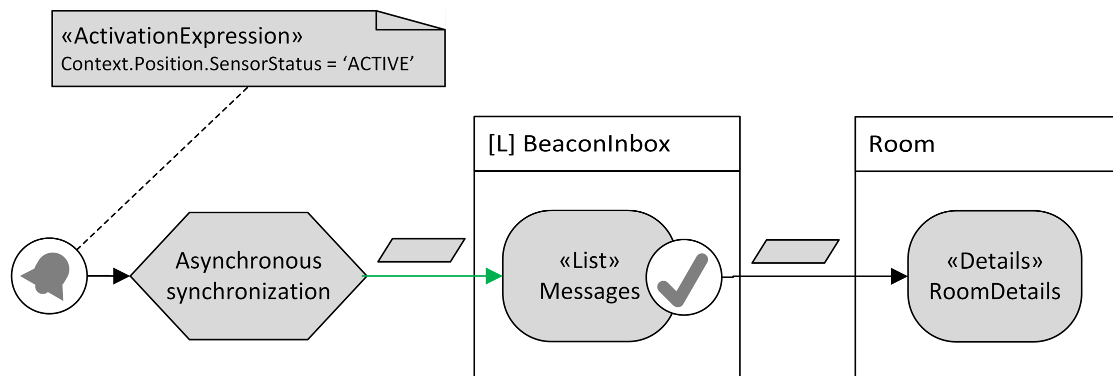


Figure 6.3: Push-Triggered Synchronization variant for beacon messages

Patterns for on-demand synchronization

Landing Screen. The landing screen contains elements based on "quasi-static" data, which is very unlikely to change over time. This data, together with other textual data of the same nature used by the application, should be synchronized at launch time (if not already stored, of course). This scenario perfectly fits with the Application Launch Synchronization pattern, in its asynchronous variant: its usage is demonstrated in figure 6.4 on the next page. Under a transfer perspective, let us underscore that if data synchronized at this stage should actually change very rarely, full transfer could be applied. Matter of fact, run differential comparisons would only increase the process' duration.

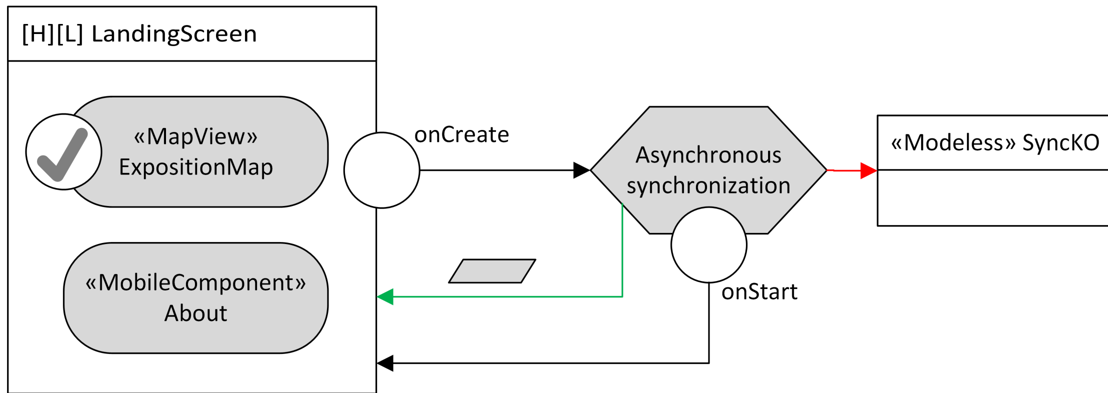


Figure 6.4: Application Launch Asynchronous Synchronization

Screens to be populated on creation. Screens that are designed to host details for a determinate entity typically features heavy contents, like for example high quality pictures, or even maps. For this kind of application components, the most clever solution is often to align data on demand, specifically when their containers are instantiated. This approach can be conceived as a slight variation of the Application Launch Synchronization pattern, replacing the application launch with the creation of the screen. Figure 6.5 portraits the application of this pattern to the Exhibitor *Screen*.

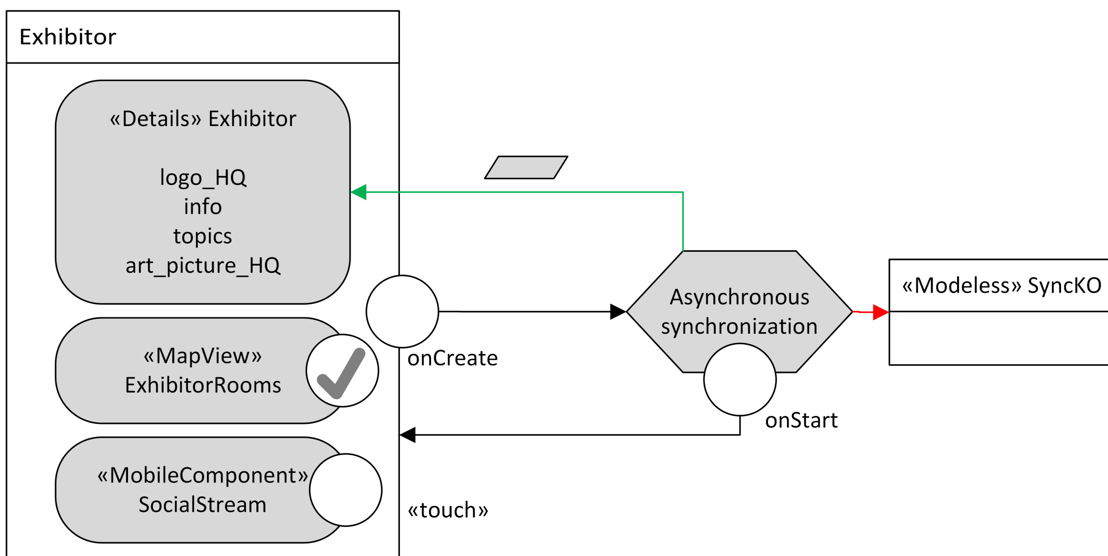


Figure 6.5: Application Launch Asynchronous Synchronization, variant for views

Manual refresh and lazy loading on real-time streams. The sections for social stream and news are two relevant components of the "Vocabolario" application and, in the meantime, feature the most dynamic data. In particular, the contents for the social stream views is continuously generated at a high pace, so that a mechanism for real-time synchronization is a requirement. Anyway, this approach needs at least an alternative and a coordinated implementation: specifically, a manually triggered refresh may act as a surrogate, while the association of content scrolling to synchronization can be integrated. Figure 6.6 depicts the dual application of Content Scrolling Asynchronous Synchronization and Pull To Refresh Asynchronous Synchronization, exemplified on the News *Screen*.

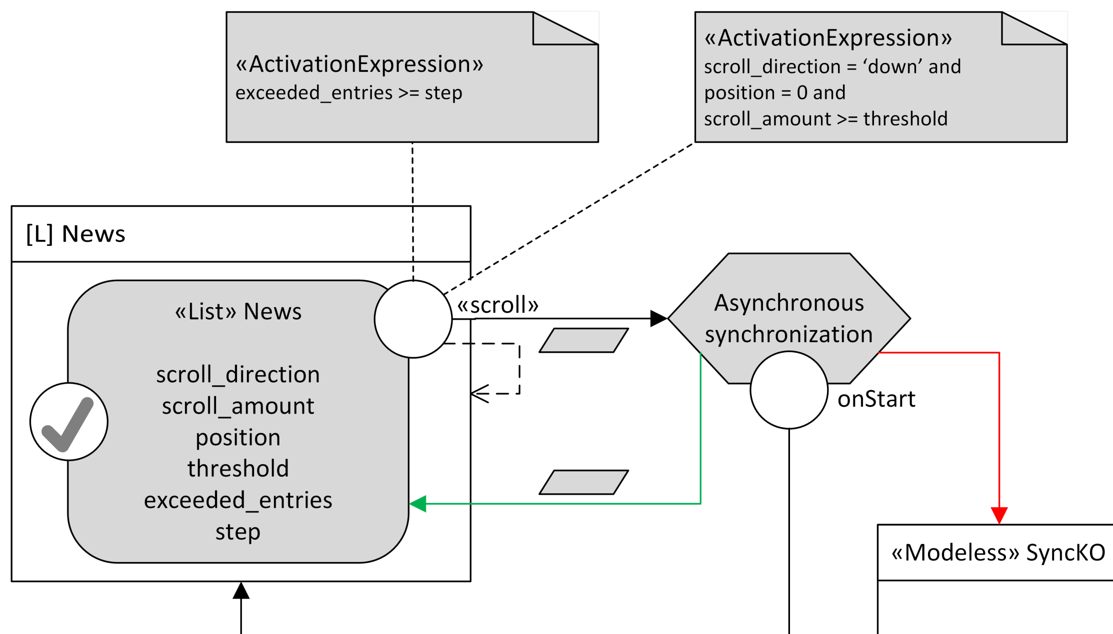


Figure 6.6: Combination of Content Scrolling and Pull-To-Refresh asynchronous synchronizations for the News view

Screens featuring forms. The components of the application featuring actions on data, associated to the submission of forms, the adoption of the Form Submission Synchronous Synchronization is the most straightforward consequence. In the case of event and agenda screens, shown in figure 6.7 on the facing page, the pattern is also exploited by the *delete* event accessible from the *ToolBar* featured in the agenda *Screen*, to permit remote removal of saved events.

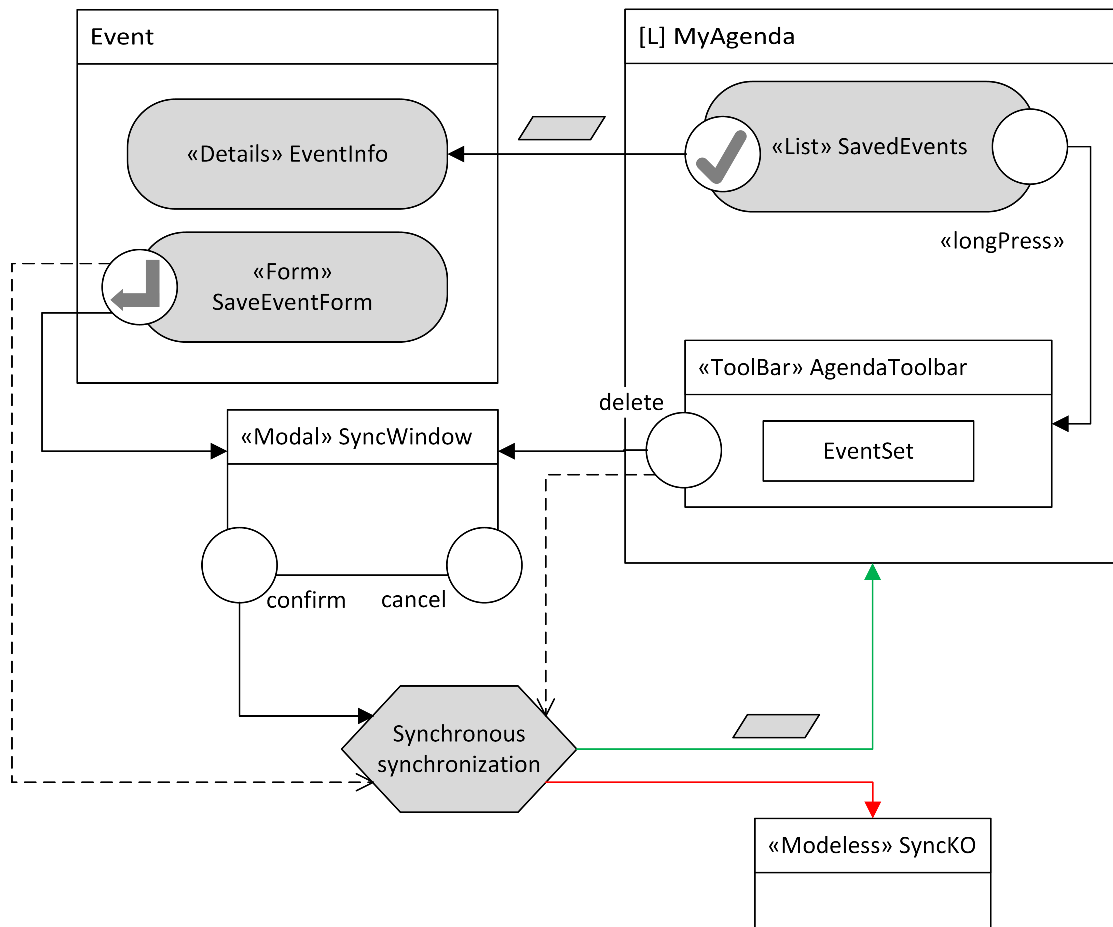


Figure 6.7: Form Submission Synchronous Synchronization applied to Event and Agenda views

6.2.3 Resulting Complete Model

Having discussed and illustrated the application of mobile data synchronization patterns to the application Front-end model, the next step is to evaluate the outcome. Figures 6.8 on the next page and 6.9 on page 95 are composing the complete IFML Front-end model of the "Vocabolario" application, now integrating the data synchronization logic. The outcome evaluation will be covered in the subsequent paragraphs.

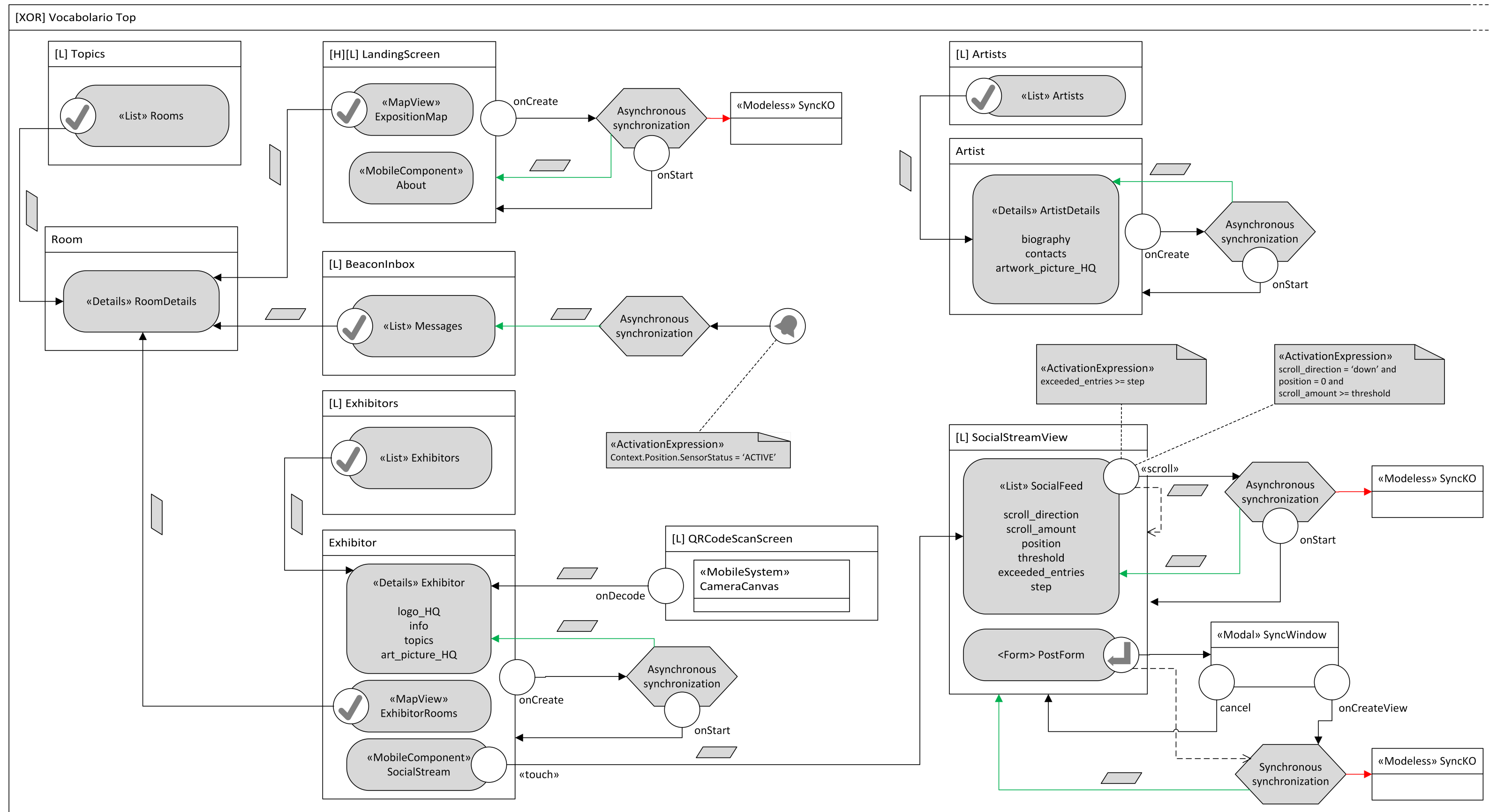


Figure 6.8: Application Front-end model, now integrating mobile data synchronization patterns - part 1

Figure 6.9: Application Front-end model, now integrating mobile data synchronization patterns - part 2

With respect to the original model, this final representation of the application's front-end is informative about the interactional, directional and temporal aspects of data synchronization mechanisms to associate to each component relying on remote data access. Further information on the nature of data, storage and transfer strategy may be inferred by analyzing the model.

In the conclusive chapter, we try to identify some more general conclusions on the introduction of the patterns we presented in this work.

Chapter 7

Conclusions and Future Work

We started our work with the specific purpose of deepening our knowledge on the approaches to data synchronization in mobile applications. The observations we made, the aspects that we emphasized and the ones that we discarded have shaped the artifacts we obtained and applied in the final stages of the thesis. Due to the experimental nature of this path, we cannot claim the goodness of our outcomes: for instance, the bottom-up approach used to analyze real-world application has some weaknesses, among which the unavailability of actual code and models. Therefore, all the considerations done on the deduced implementations are to be deemed as mere reasoned hypothesis. In Chapter 6, we have applied the patterns we introduced, but their advantages, liabilities and possible implied compromises are still to be evaluated. The evaluation process should involve a decent amount of modeling scenarios, also to study possible extensions or improvements to clear the idiosyncrasies of our modeling artifacts. On the sidelines of this demonstrative experience, in the next paragraphs, we try to abstract some positive aspects and possible drawbacks that can be inferred by the examination of the outcome. Finally, we attempt to identify some future steps to continue the work we have been presenting.

Benefits

- Gain in terms of expressive power of the model, which is able to give a remarkable amount of information about synchronization mechanisms, in particular on how they affect the user interface. Considering the high amount of *MobileComponents* which are directly bound to data synchronization in client-server applications, like implicit and explicit interactors, this benefit acquires even more relevance.
- The application of the patterns breaks the barriers between front-end and back-end design, increasing the continuity of modeling tasks.

- Patterns are easy to combine and modify, to adapt them to particular scenarios. This has been widely demonstrated in the previous chapter, by applying variant of the presented patterns to several components of the modeled application.

Liabilities

- Data synchronization model is not complete, as information inferred by the front-end model is obviously coarse-grained. On the other hand, this issue is not unexpected, being data alignment a diagonal problem with respect to front-end and back-end logic.
- Some notions on the transfer policies, storage strategies and dynamic properties of data are only partially deducible. In the worst case, we have no information at all about those aspects.
- Some constraints of the modeling language, IFML, reduce the intuitiveness of the patterns: for instance, the restitution of the focus to the calling view-container is represented as a flow outgoing from an "on start" event (associated to the synchronization action). This is clearly a workaround to achieve a coherent representation.

Future Work

In this thesis we have been focusing on the enhancement of models with a pattern-based approach but, as already observed, the model realization is just a stage in Model Driven Engineering. The next step to make our effort more valuable would be the definition of rules and templates for code generation (for each target mobile platform). This stage would exploit the *implementability* of the IFML language, and should ideally represent the final step to evaluate the effectiveness of our introductions.

Chapter 8

Related Work

8.1 Design Patterns in Software Development

The concept of "Design Pattern" has been playing a central role in the study presented in this research work. Despite of this fact, so far we have never deepened the concept of pattern within the environment it for which it was intended, that is software design.

In simple words, a pattern highlights a problem occurring multiple times in a given environment and describes the solution to that problem, in such a way that the solution can be re-used over and over. More specifically, in software industry a design pattern is described by a written formal document detailing the elements characterizing it. According to [8], authored by the so called "Gang of Four", which in this discipline is a fundamental reference document, the headings of the specification must include Intent, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses and Related Patterns. Eventually, as design patterns have been growing in popularity, alternative pattern forms have been defined: the most common are Portland, Coplien (adopted to introduce patterns in chapters 2, 4 and 5), POSA and P of EAA.

In the upcoming paragraphs (Naming, Forces, Issues) we are reporting some remarks on patterns creation, taking inspiration from the considerations articulated by the author on enterprise software Martin Fowler in [3].

Naming. A great benefit resulting by the identification of patterns is given by the possibility to organize, name and catalog the problem-solution scenarios, so that their reusing potential is facilitated. For this reason, choosing an effective, evocative name is a valuable task, keeping in mind that it should be part of a vocabulary of software techniques.

Forces. A common misunderstanding about patterns results from thinking that they should introduce something new in software engineering. This not true at all, as their goal is to capture knowledge, rather than invent it. Precisely because of this, when presenting a pattern, writers should strongly motivate their need, emphasizing the forces behind them. Under this perspective, trying to think about occasions in which the pattern usage would not be recommended can be useful, at least to identify an alternative pattern.

Issues. Granularity express the surface of conceptual ground covered by a pattern. Typically, selecting granularity opens a debate not allowing absolute solutions: in fact, deciding where to place boundaries between different patterns operating on nuances of the same problem is a hard task. Another critical aspect is the task-oriented nature of patterns. Tool orientation in software leads to simplifications, and pattern writing does not represent an exception: observing a framework and identifying tools is intuitive. But patterns should stay task-oriented, because of the natural answer that task orientation provides to designers adopting the pattern.

The closing paragraph of this section tries to connect the patterns introduced in this document to classic object-oriented design patterns —as classified in [8]— based on their problem-solving approach.

The Behavioral Approach. By definition, "behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects" [8, p. 249]. They focus not just on objects and classes, zooming also on the existing communication mechanisms between them. This kind of approach broadly resembles the one used to create the patterns illustrated in our research: even though it may appear tough to visualize through the IFML renditions, this similarity is emphasized by the UML sequence diagrams. For instance, by looking at fig. 5.6 on page 70, representing the dynamics of "Content Scrolling Asynchronous Synchronization", it is deducible that peer objects cooperate to finalize a task being loosely coupled. Specifically, the "onScroll" event may be handled by different objects, as well as the request sent to the synchronization service can originate from more than a source. Even synchronization itself can be handled by several actors: most of these responsibilities depend on framework or even system implementation design. Processes of this nature are similar to the scenarios portrayed by the "Chain of Responsibility" classic pattern [8, p. 251], in which requests are forwarded and handled with flexibility, and objects collaborating are loosely coupled.

8.2 MDE, IFML and Mobile Applications

Besides the notion of pattern, the other pillar of our study is represented by the scope in which the exploitation of patterns takes place: the modeling environment.

The MDE approach. Model Driven Engineering is an approach to development using models as main artifacts. This translate to the exploitation of models during the whole process of development, in contrast with a normal approach that typically takes advantage of their features only in the analysis and design phases.

The relevance of MDE in software engineering has known a notable growth in the past five years thanks to rising forces like application pervasiveness and their multi-platform, distributed nature [5, p.71]. The adoption of an MDE approach eludes the weaknesses of a trial-and-error typical of adaptively fixed systems, fostering the focus change on gathering abstract representations of the knowledge governing application domains and their re-use.
On the other hand, code generation from software models is a hard task, since it requires a complete understanding of the model to code transformation semantics. As a consequence, end-to-end generation of application code matching the level of quality of a hand-crafted solution remains an ambitious goal. Despite these difficulties, some impressive outcomes has been achieved, like in the case of WebRatio [2], company in the MDE tool market, as reported in [5, p.79].

The central role of models in MDE products and services made the industry agree on the idea of setting some standards to produce platform-independent models. As already mentioned in chapter 2, the OMG (Object Management Group) adopted the Interaction Flow Modeling Language (IFML) as a standard in July 2014.

IFML and Mobile Applications Extensions. Let us now explain how IFML contributes to create suitable platform-independent models (PIM) of graphical user interfaces for application deployed on multiple systems. As explained in the details in [6], an IFML PIM provides:

- the specifications of the view composition and content, respectively illustrating the schema of containers and their components;
- the specifications of events (commands) affecting the state of the User Interface and of the transitions (effects of interaction) portraying the effects implied by the state change;
- the parameter binding specification, listing the dependencies between components and components and actions;
- the actions reference, specified by interaction flows connecting events to affected view containers or components.

The PIM should be designed for change and its design should adhere to a set of "golden rules" of the language [4]:

- it should be *concise*, conveying the front-end model in a single diagram and avoiding redundancy by exploiting *inference from the model*;
- it should encourage *extensibility*, allowing the adaptation to novel requirements, interaction modalities;
- it should be *implementable*, to ensure that models can be mapped easily into executable applications;
- *not everything should be in the model*, meaning that, for instance, presentation aspects should not be covered.

In particular, the extensibility property of the language is what we relied on to conduct our research work. In fact, IFML provides tools for defining novel concepts and interaction paradigms, possibly substantially different from the ones for which the language has been designed. Under this perspective, our study demonstrated this concept by integrating in models information on data synchronization, an aspect that is typically closer to back-end logic, rather than front-end design.

To conclude this brief overview of the features of IFML that enabled this work, we mention some extensions —widely adopted for the representation of the patterns presented in the previous chapters— of the language conceived for the design of mobile applications.

The class *Screen*, representing a screen of the application, has been defined by extending *ViewContainer*, as well as *ToolBar*, which may contain other containers and have on its boundary a set of events. The *MobileSystem* stereotype has been introduced to distinguish ViewContainers that in mobile interfaces are devoted to specific functionalities (like Notifications, Settings). *MobileComponent* extends the classic IFML class *ViewComponent* and denotes mobile view components, such as buttons or icons.

Bibliography

[1] jimcoplien - gertrudandcope. `http://sites.google.com/a/gertrudandcope.com/www/jimcoplien/`. Accessed: 2015-04-28. 19

[2] Rapid mobile app and web application development platform. `http://www.webratio.com/`. Accessed: 2015-05-05. 18, 101

[3] Writing software patterns. `http://www.martinfowler.com/articles/writingPatterns.html`. Accessed: 2015-06-20. 99

[4] Marco Brambilla and Piero Fraternali. *Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML*. Morgan Kaufmann, 2014. 18, 102

[5] Marco Brambilla and Piero Fraternali. Large-scale model-driven engineering of web user interaction: The webml and webratio experience. *Science of Computer Programming*, 89:71–87, 2014. 101

[6] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. Extending the interaction flow modeling language (ifml) for model driven development of mobile applications front end. In *Mobile Web Information Systems*, pages 176–191. Springer, 2014. 18, 102

[7] Andre Charland and Brian Leroux. Mobile application development: web vs. native. *Communications of the ACM*, 54(5):49–53, 2011. 17

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994. 99, 100

[9] Eeva Kangas and Timo Kinnunen. Applying user-centered design to mobile application development. *Commun. ACM*, 48(7):55–59, July 2005. 17

[10] Drew McCormack. Data synchronization. `http://www.objc.io/issue-10/data-synchronization.html`, March 2014. Accessed: 2015-05-12. 30, 31

[11] Zach McCormick and Douglas C Schmidt. Data synchronization patterns in mobile application design. 2012. 37, 38