Capitolo 1

Architettura

1.1 Introduzione

Per poter gestire meglio lo sviluppo l'intero sistema è stato suddiviso in due parti, visibili concettualmente e concretizzate in *solution* (soluzioni) così definite da Visual Studio. Esse sono:

- Updater Client;
- Updater Server.

Ad esse sono stati associati i vari progetti definiti all'interno di namespace:

Updater: racchiude le componenti principali del programma che fornisce le funzionalità di installazione e aggiornamento dei prodotti;

UIControls: raggruppa le componenti grafiche che seguono il pattern MVVM (quindi per ogni componente grafica, view, abbiamo un corrispondente ViewModel e Model);

UpdaterLibrary: contiene le componenti che incapsulano una corretta comunicazione con il servizio REST e effettuano le operazioni per scaricare, installare e aggiornare un prodotto software;

UpdaterSerialization: contiene le componenti che racchiudono i dati trasmessi tra server e client, quindi serializzati.

UpdaterServer: contiene le componenti necessarie per il funzionamento del servizio RESTful;

UpdaterDBEF: (DBEF = Database Entity Framework) contiene le componenti che implementano il pattern ORM il quale consente una facile relazione tra le componenti orientate agli oggetti e il database relazionale.

Di seguito l'associazione delle varie macro componenti con le solution:

Updater Client composto dalle macro componenti:

- Updater;
- UpdaterUIControls;
- UpdaterClientLibrary;
- UpdaterSerialization.

Updater Server composto dalle macro componenti:

- UpdaterSerialization;
- UpdaterServer;
- UpdaterDBEF.

Come è possibile notare, le componenti in UpdaterSerialization permettono lo scambio di informazioni tra client e server.

1.2 Pattern Architetturali

L'intero sistema segue una struttura a livelli (figura 1.1) in cui la dipendenza tra le macro componenti segue un flusso verso il basso.

La comunicazione avviene principalmente solo tra macro componenti vicine. In particolare la macro componente Updater Client Library rende indipendente le altre componenti client dalla comunicazione attraverso la rete e l'uso delle richieste HTTP.

Per la parte client si è seguito il pattern Model View ViewModel (MVVM) incoraggiato dalla tecnologia WPF e rafforzato dall'uso del framework di supporto Catel. Quest'ultimo mette a disposizione tag, attributi e classi che permettono una gestione più semplice delle tre parti: View, ViewModel e Model. Ogni componente grafico avrà il suo corrispettivo ViewModel (logica e dati utili al componente View) e se necessario anche il Model (se in possesso di dati persistenti, per esempio prelevati dal database).

Per la parte server invece si utilizza il framework ASP.NET il quale attraverso la definizione di classi **controller** permette la costruzione di un servizio RESTful. La comunicazione tra Web API e database avviene grazie ad Entity Framework il quale implementa il la tecnica di programmazione Object-Relational Mapping (ORM).

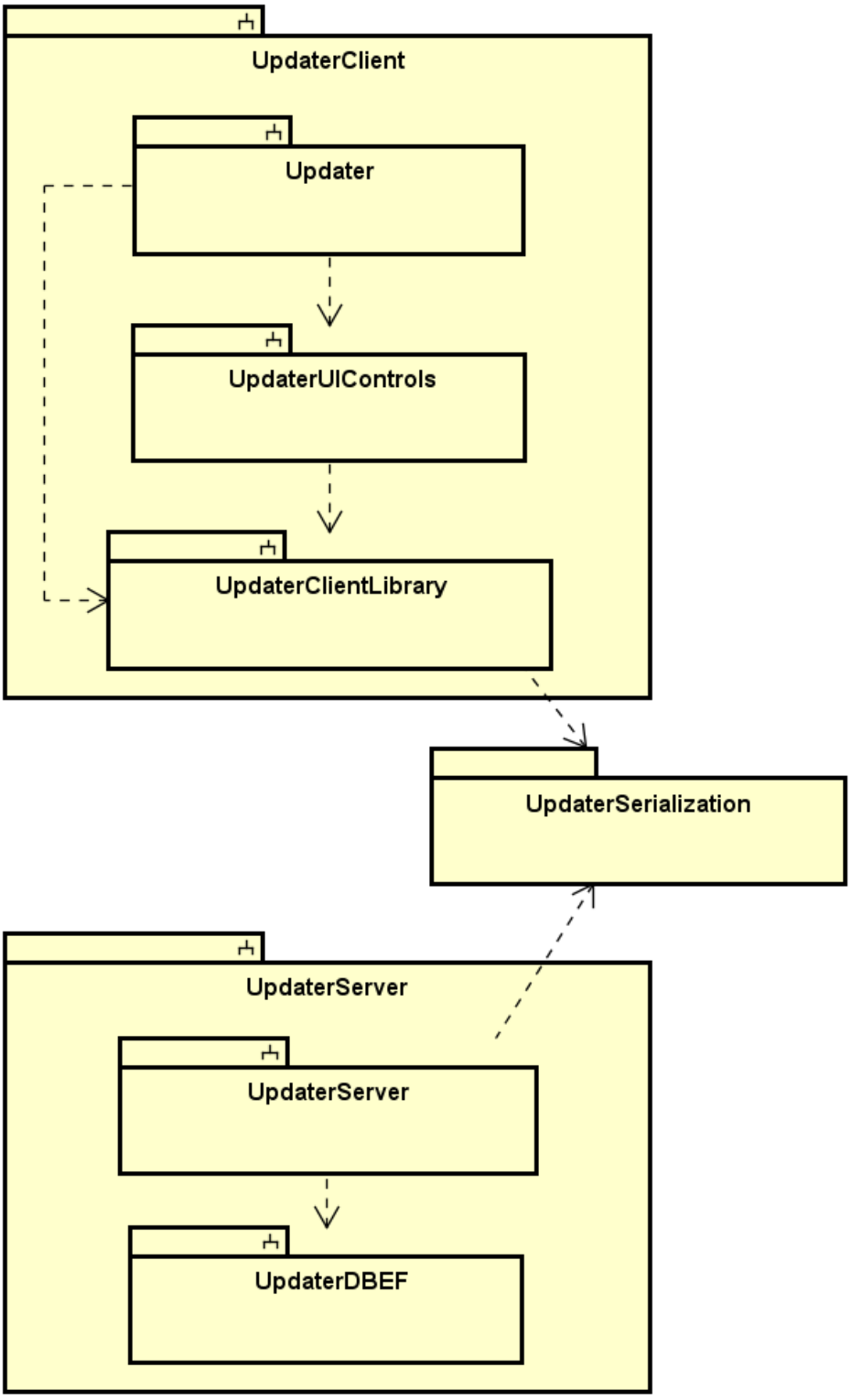

Figura 1.1: Dipendenze tra macro componenti del sistema Updater

1.3 Macro componenti

Nella presente sezione vengono descritte in dettaglio le macro componenti riportando per ciascuna una descrizione esauriente e la struttura con le componenti interne e le relazioni tra esse. Dove necessario vengono discusse le scelte e le possibili criticità cercando di proporre soluzioni migliori e migliorie che in futuro potrebbero essere implementate.

1.3.1 Updater

1.3.1.1 Descrizione

Identifica un programma eseguibile lato client composto principalmente da un interfaccia grafica per agevolare l'utente nel download, nell'installazione o nell'aggiornamento dei prodotti software e relativi prerequisiti.

1.3.1.2 Responsabilità

Responsabilità del namespace:

- Gestire l'avvio dell'applicazione WPF;
- Gestire eventuali parametri all'avvio dell'applicazione;
- Gestire la finestra principale dell'applicazione;
- Gestire la finestra delle impostazioni dell'applicazione;
- Gestire l'icona di notifica nella TaskBar;

1.3.1.3 Implementazione

Dipendenze esterne

- WPF;
- Catel;
- MahApps;
- Command Line Parser.

Dipendenze interne

- Salvagnini.UICatelControls;
- Salvagnini.UIControls.

Il programma si basa sul framework Windows Presentation Foundation il quale mette a disposizione la classe `Application` singleton e primo oggetto istanziato all'esecuzione del programma. `Application` è quindi stata ereditata dalla classe `App` la quale è responsabile di:

- creare la finestra principale e la schermata secondaria delle impostazioni utente;
- creare la Notify Icon che comparirà nella barra dello start (*TaskBar*) e della comparsa;
- creare avvisi (*baloon*) in risposta al completamento corretto o in errore delle operazioni automatiche.

La finestra principale è identificata dalla classe `MainWindows` estende la classe `MetroWindow` distribuita dall'UI toolkit *MahApp* il quale rende la finestra in stile alle moderne applicazioni windows aderenti al metro design. Una finestra secondaria è rappresentata dalla classe `SettingsView` la quale si occupa di gestire le componenti grafiche per le preferenze utente dell'intera applicazione, tali preferenze (o impostazioni) impatteranno nel funzionamento della `TaskBarIcon`. In se il progetto è costituito solo dalla finestra principale e dalla gestione dell'icona nella taskbar del sistema contiene quindi prettamente componenti grafiche e la logica di base di esse e dell'avvio dell'applicazione (implicito grazie all'uso di `Application`.

1.3.1.4 Funzionamento

La struttura segue il pattern MVVM come si vede in figura . `App` è una classe singleton resa disponibile dal framework WPF, da questa il programma inizializza la `MainWindow` e tutte le componenti grafiche (View) coinvolte insieme ai rispettivi ViewModel e Model. Le classi interne al namespace `Views` sono costituite da una parte descritta in XAML (descrive l'aspetto prettamente grafico) e una in C# (il *code behind*). La classe `Options` usufruisce della classe `IParserState` della libreria Command Line Parser consentendo all'applicazione di essere eseguita con alcuni parametri utili per gli eventuali riavvi necessari nell'installazione di qualche prodotto.

1.3.1.5 Criticità e possibili miglioramenti

Il pattern architetturale MVVM garantisce una certa separazione tra le componenti e l'uso del framework Catel rende le comunicazioni tra View e ViewModel molto chiare e semplici, anche tra diversi ViewModel grazie all'attributo `InterestedIn`. L'uso della TaskBarIcon costringe all'inserimento di codice di logica (e quindi non riguardante la grafica) nel *code behind* della classe `App` ciò oltre a non essere buona pratica dà una responsabilità di più

basso livello (creazione dei baloon) rispetto alla creazione delle finestre della classe `App`. [1]

Nel caso in cui ci sia la necessità di aggiungere nuove responsabilità per la classe `App` è necessario un redesign per spostare il codice che coinvolge la classe `TaskBarIcon` ad una nuova classe.

[1] Responsabilità di diverso livello rompono il *principio di simmetria* individuato da Kent Beck in *Implementation Pattern, Cap. 3 - pag. 15*

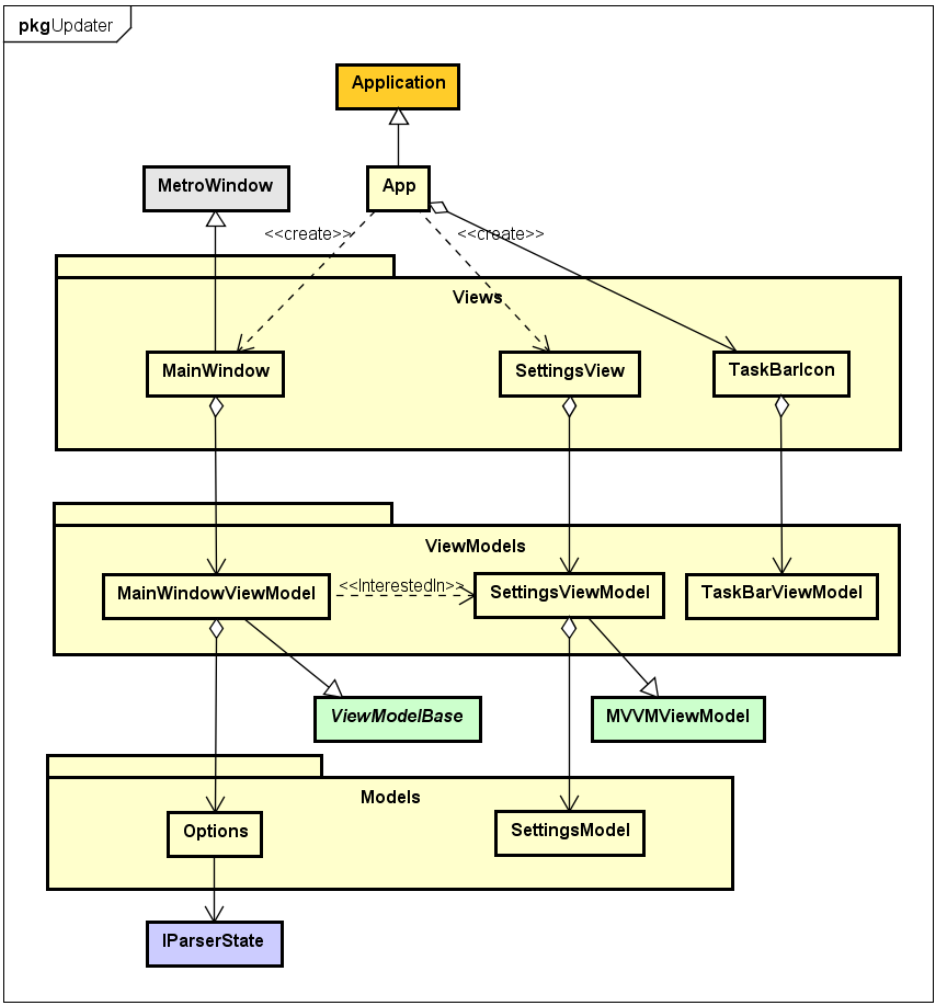

Figura 1.2: Struttura macro componente Updater

1.3.2 Updater User Interface Controls

1.3.2.1 Descrizione

Identifica un insieme di componenti grafici utilizzabili da software esterno per costruire un'interfaccia grafica. Tali componenti sono utilizzate da Updater e potranno essere utilizzate da altri prodotti software.

1.3.2.2 Responsabilità

- Fornire le componenti grafiche principali che consentono all'utente di visualizzare, scaricare, installare, reinstallare e aggiornare un prodotto software.

1.3.2.3 Implementazione

Dipendenze esterne

- WPF;
- Catel;
- MahApps.

Dipendenze interne

- Salvagnini.UICatelControls;
- Salvagnini.UIControls.

La macro componente è costituita da tre elementi grafici principali:

1. ProductsPanel;
2. ProductControl;
3. ProductInfo.

Il primo rappresenta una lista degli elementi grafici `ProductControl` e offre vari filtri da applicare ad essa compresa una ricerca per nome. Il secondo rappresenta un singolo prodotto software e fornisce i pulsanti per effettuare le operazioni di installazione, reinstallazione, download e aggiornamento. Il terzo è legato al primo con l'attributo offerto da Catel *InterestedIn*, in base alla selezione di un elemento della lista questo mostra informazioni più dettagliate su di esso. Tutti gli elementi implementano il pattern MVVM pertanto sono scomposti in tre parti. La parte prettamente visiva (*View*) è costruita in XAML e C#.

1.3.2.4 Funzionamento

WPF instanzia le classi `View`. `ProductInfoView` istanzia il proprio `Model`, ossia `ProductInfoViewModel`.

Per `ProductsPanelView` invece il comportamento è diverso (figura), Se il proprio `ProductsPanelViewModel` non è già istanziato prima costruisce `ProductsPanelModel`, poi `ProductsPanelViewModel`. Dopo la creazione di `ProductsPanelModel`, `ProductsPanelViewModel` incarica `ProductsPanelModel` di eseguire una richiesta http per recuperare le informazioni sui prodotti software.

Viene quindi istanziato un `ProductControlModel` per ogni prodotto software. Terminato questa operazione `ProductsPanelViewModel` può istanziare per ogni `ProductControlModel` un rispettivo `ProductControlViewModel`. Infine grazie al meccanismo del `DataContext` di WPF la classe `ProductControlView` può recuperare il rispettivo `ProductControlViewModel`.

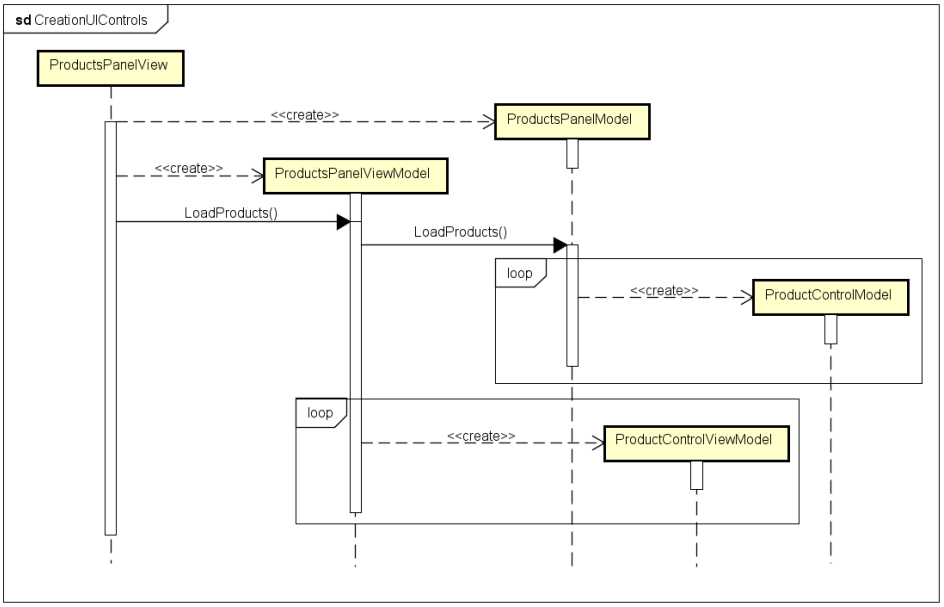

Figura 1.3: Diagramma di sequenza per la costruzione degli elementi grafici UIControls

1.3.2.5 Criticità e possibili miglioramenti

Il macro componente presenta confusione nell'uso di `DataContext`, responsabilità un po' confuse dovute e risulta necessario un redesign. Infatti prima vengono costruito i vari `ProductControlModel` e in seguito i `ProductControlViewModel` mancando di coesione e incrementando la complessità della struttura. Tutto

ciò rende ancor più difficile la comprensione delle componenti e quindi la modifica futura.

La classe `ProductControlModel` racchiude in sé logica delle componenti di *Updater Client Library* poiché all'interno di queste operazioni sono necessari dei feedback che impattano nell'interfaccia grafica, inoltre è possibile che durante l'operazione sia richiesto il riavvio dell'app. Un'alternativa forse più valida sposterebbe la logica nella libreria definendo la possibilità che essa lanci degli eventi indicanti lo stato delle operazioni e a questi `ProductControlModel` reagisca opportunamente. Questa alternativa non è stata implementata per evitare la complessità nella gestione della concorrenza. Si verrebbe infatti a creare una situazione in cui due differenti thread devono comunicare tra di loro.

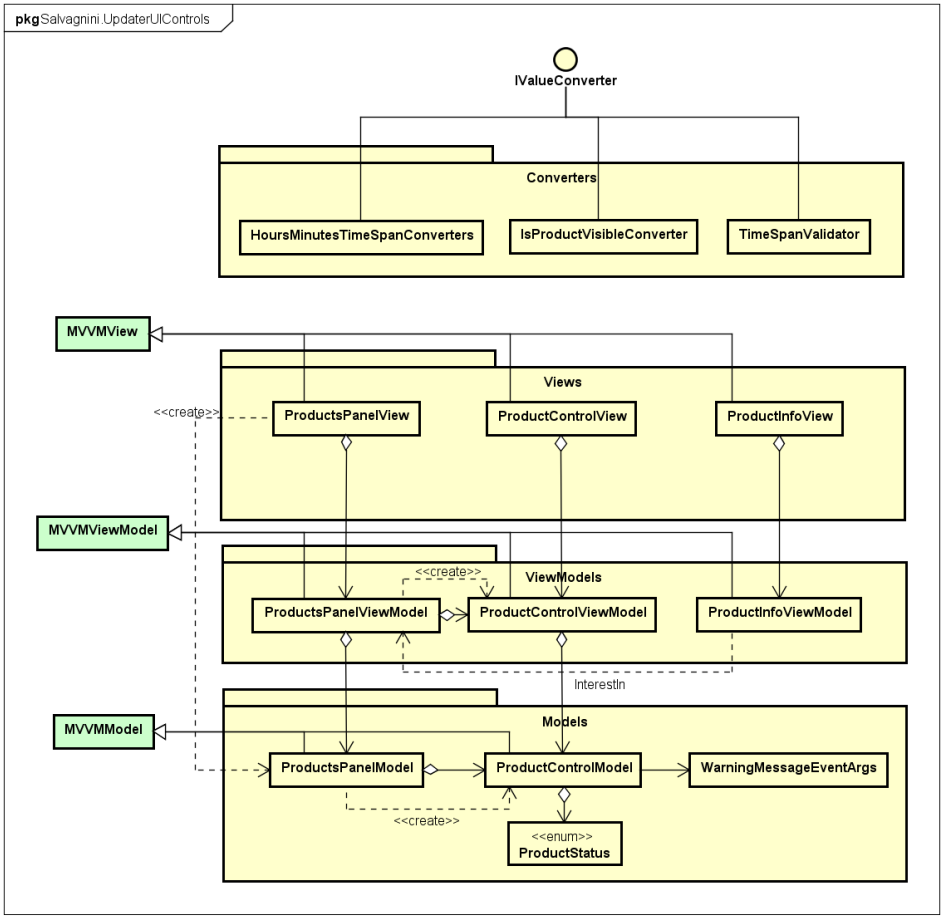

Figura 1.4: Struttura macro componente Updater UIControls

1.3.3 Updater Client Library

1.3.3.1 Descrizione

Identifica un insieme di componenti software incaricate di effettuare delle operazioni di controllo locali (per es. presenza dei programmi Salvagnini) e di gestire le comunicazioni con la Web API tramite servizio REST e il formato dati JSON. Tali componenti sono utilizzate dalle macro componenti Updater e UpdaterUIControls. Sono state rese indipendenti per poter essere utilizzate da altri prodotti software.

1.3.3.2 Responsabilità

- Incapsulare la comunicazione con il servizio REST;
- Consentire le operazioni su un prodotto software:
  - Download;
  - Installazione;
  - Reinstallazione;
  - Aggiornamento;
- Consentire l'esecuzione automatica delle operazioni elencate precedentemente;
- Fornire gli strumenti per il corretto riavvio dell'applicazione;
- Fornire gli strumenti per il salvataggio su file XML delle operazioni effettuate (per es. Salvare la data di installazione di un prodotto);
-

1.3.3.3 Implementazione

Dipendenze esterne

- Catel.

Dipendenze interne

- Updater Serialization.

La libreria è costituita principalmente da tre parti:

1. Dalle componenti che si occupano di comunicare con il servizio REST e di effettuare le operazioni sui prodotti software;
2. Dalle componenti necessarie al salvataggio di dati in formato XML;

3. Dalle componenti che si occupano di automatizzare le operazioni;

La prima parte è costituita da `ProductWrapper` e `PrerequisiteWrapper` le quali rappresentano un incapsulazione dei dati deserializzati ricevuti dal servizio RESTful grazie alla classe `UpdaterService`. Le componenti Wrapper incapsulano tutte le operazioni che è possibile effettuare su un prodotto software e sui prerequisiti software. Per maggiori dettagli si rimanda alla sezione .

La seconda parte è costituita dalle classi all'interno del namespace `Models`. La classe singleton `ObjectManager` è incaricata di gestire il caricamento e il salvataggio dei dati resi persistenti in un file XML. Questi sono rappresentanti dalle classi `Set` che racchiudono una struttura dati contenente oggetti contenenti i dati effettivi da serializzare.

La terza parte è costituita da una gerarchia, le classi concrete di tale gerarchia sono incaricate di lanciare le corrette operazioni automatiche di download, installazione o aggiornamento. Questo sulla base delle preferenze impostate dall'utente e attraverso l'uso di timer. La classe `AppRestarter` racchiude la logica per il corretto riavvio dell'applicazione nel caso un'operazione automatica lo richiedesse.

1.3.3.4 Funzionamento

La libreria mette a disposizioni delle classi involucro, in particolare `ProductWrapper`, che consentono di effettuare le operazioni sui prodotti software. Queste stesse classi `Wrapper` si occupano di rendere persistenti i dati delle operazioni svolte attraverso l'uso delle componenti interne al namespace `Models`. Oltre alle classi `Wrapper` la libreria mette a disposizione le classi `AutomaticUpdates`, esse basano il proprio funzionamento in base ad un timer interno, alla loro creazione il timer parte e allo scadere la classe effettuerà l'operazione per cui è incaricata in modo completamente automatico (da qui la dipendenza tra `AutomaticUpdatesController` e `ProductWrapper`), il timer quindi riparte. Ogni qual volta il timer scade le classi `AutomaticUpdates` lanciano un evento `ProductToUpdateEventArgs` in questo modo sfruttando il pattern Observer [2] insito nel linguaggio C# è possibile reagire allo scadere del timer anche in componenti esterne.

1.3.3.5 Criticità e possibili miglioramenti

Dipendenza circolare con ProductWrapper e UpdateService, a causa della scelta della lazy initialization per i PrerequisiteWrapper. UpdaterService potrebbe contenere metodi statici per risolvere. La lazy initialization potrebbe essere eliminata per rimuove la dipendenza da PrerequisiteWrapper con UpdaterService. Manca qualche namespace che organizzerebbe meglio la

[2] GoF - Design Patterns: Elements of Reusable Object-Oriented Software

libreria, ad esempio AutomaticUpdates potrebbe essere un nuovo namespace per la gerarchia di classi strettamente connesse. ProductWrapper contiene codice duplicato con AppRestarter come conseguenza di un refactoring, oltre ad assumersi un numero elevato di responsabilità che suggerisce di spezzare la classe in più classi. La scelta di contenere tutti in una è data dal fatto che le operazioni necessari erano ben definite (download, install e update) e si suppone che in futuro non ne servano altre. Nel caso contrario si potrebbe pensare a rappresentare ogni operazione come una classe implementando così il Command pattern [3]

Le componenti incaricate della deserializzazione e serializzazione dal file XML sono spesso ridondanti, l'uso di classi generiche consentirebbe un'ulteriore flessibilità al sistema evitando di dover ad ogni aggiunta di dati da rendere persistenti la codifica di ulteriori classi praticamente identiche. L'uso di classi generiche consentirebbe anche al signleton `ObjectsManager` di acquisire maggior flessibilità. Attualmente questa operazione non è stata fatta per limiti tecnologici e conoscitivi del framework Catel, si è sperimentato infatti che non è possibile rendere generica una classe che estende `ModelBase`, probabilmente il limite deriva dal fatto che i tipi parametrici contano solo nella precompilazione, di fatto poi tutto è tipizzato con la classe `Object` di C# in questo modo il meccanismo di serializzazione non può più distinguere una classe generica istanziata con un tipo da un'altra istanziata con tipo diverso.

[3] GoF - Design Patterns: Elements of Reusable Object-Oriented Software

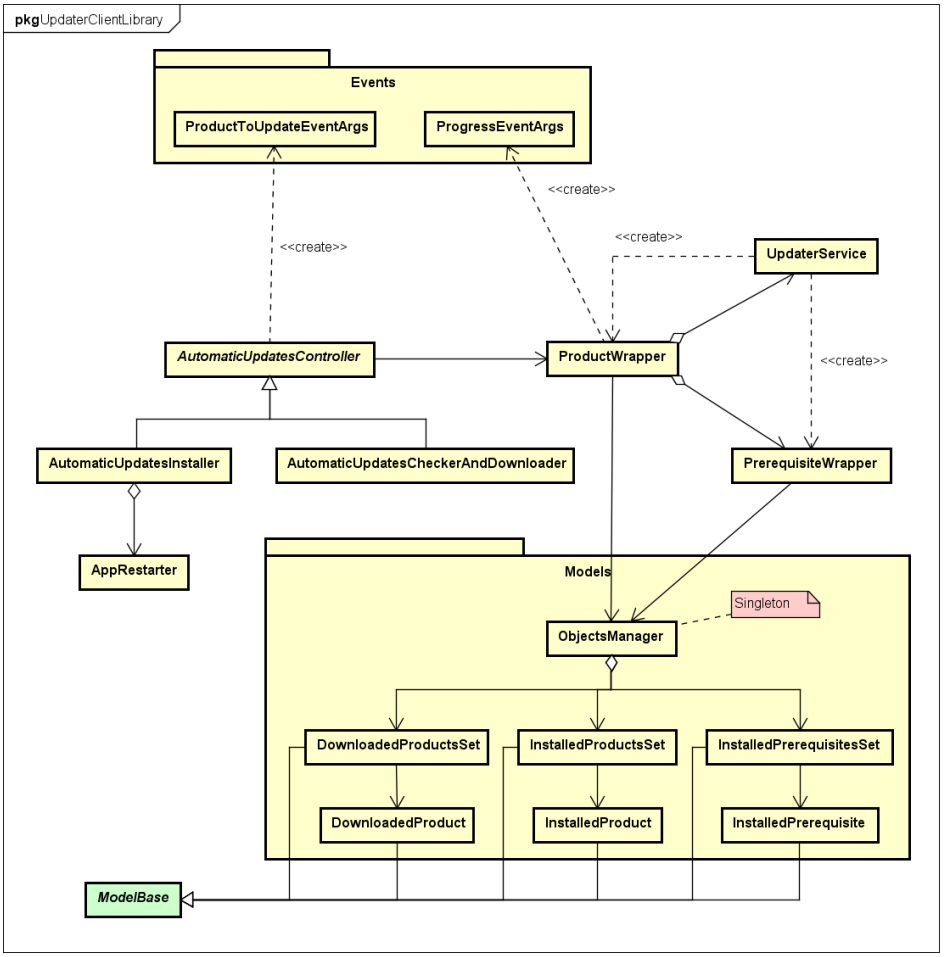

Figura 1.5: Struttura macro componente Updater Client Library

1.3.4 Updater Serialization

1.3.4.1 Descrizione

Identifica l'insieme di definizioni di oggetti che permettono la trasmissione di dati in una rete tra server e client tramite serializzazione.

1.3.4.2 Responsabilità

- Definire oggetti serializzabili contenenti dati.

1.3.4.3 Implementazione

Dipendenze esterne Nessuna.

Dipendenze interne Nessuna.

Le classi che terminano con il suffisso `Object` sono definizioni di oggetti che contengono solo campi dati e costruttori. I campi dati sono etichettati con l'attributo `[DataMember]` in tal modo il framework .NET è capace di trasformarle in formato JSON così da poter trasmettere informazioni attraverso una rete. La classe `Serializer` si occupa di facilitare tale serializzazione attraverso la definizione di metodi statici per la lettura e scrittura di dati serializzati.

1.3.4.4 Funzionamento

L'unica classe del namespace che racchiude logica è `Serializer`, la quale semplicemente semplifica la serializzazione e la deserializzazione definendo due metodi statici. Le altre componenti non racchiudono logica al loro interno. Nessuna componente ha dipendenze esterne per questo motivo si omette il diagramma della struttura del macro componente.

1.3.4.5 Criticità e possibili miglioramenti

L'unico punto critico sono i costruttori delle classi `Object` serializzabili. Nel caso infatti si dovesse aggiungere un campo dati si è costretti a definire un nuovo costruttore e definire un valore di default per il campo dati aggiunto. Nel caso questo non possa avvenire, tutte le dipendenze verso la classe necessiterebbero di essere modificate. Per far sì che questo sforzo di modifiche sia limitato le dipendenze verso questo macro componente sono state limitate **esclusivamente** macro componenti *Updater Client Libary* e *Updater Server* e così dovrebbe essere mantenuto in future estensioni.

1.3.5 Updater Server

1.3.5.1 Descrizione

Identifica il server con sistema IIS su cui opera la web API supportata da un database relazionale che implementerà un servizio REST col quale comunicheranno il macro componente Updater Client Library e le pagine web amministrative del database).

1.3.5.2 Responsabilità

- Fornire un servizio REST per trasmettere i dati dei prodotti software e dei relativi prerequisiti software prelevati da un database;
- Fornire delle pagine web amministrative per facilitare le modifiche ai dati contenuti nel database;

1.3.5.3 Implementazione

Dipendenze esterne

- ASP.NET;
- JQuery;
- Bootstrap;
- KnockoutJS.

Dipendenze interne

- Updater Serialization;
- Updater Entity Framework Database (EFDB).

Design patter Il macro componente è stato sviluppato seguendo la struttura imposta dal framework ASP.NET.

Per la parte server del servizio REST è stata impostata la configurazione del servizio nelle componenti del namespace `App_Start` preconfigurate dal framework e per richiedere l'autenticazione alle richieste http è stata creata la classe `BasicAuthenticationHandler`. Le componenti principali sviluppate sono le classi `Controller` ossia classi i cui metodi vengono invocati in base alla richiesta Http ricevuta. Le gerarchie `Putter` e `Poster` sono state sviluppate con lo scopo di supportare tali metodi. Il namespace `SerializableFactory` racchiude classi `Factory` [4] le quali racchiudono metodi statici per la creazione di oggetti `Object` definiti in *Updater Serialization* a

[4] GoF - Design Patterns: Elements of Reusable Object-Oriented Software

partire dagli oggetti distribuiti da *UpdaterDBEF*. Per quanto riguarda la parte web (omessa nel diagramma in figura) invece sono state create tre pagine web in html5 decorate con l'aiuto del framework Bootstrap. Le pagine web interagiscono con il servizio REST tramite l'uso della tecnica Asynchronous JavaScript and XML (AJAX) resa disponibile dalla libreria JQuery.

1.3.5.4 Funzionamento

Una volta avviata la web api vengono caricate le dovute impostazioni codificate nelle classi all'interno di `App_Start`. A questo punto ad ogni richiesta http inviata al server agisce la classe `BasicAuthenticationHandler` che verifica la presenza di nome utente e password corretti nella richiesta. Una volta accertata la richiesta il flusso del programma è passato alle classi `Controller`, qui viene eseguito il codice del metodo che corrisponde all'uri della richiesta.

1.3.5.5 Criticità e possibili miglioramenti

Attualmente molti dati e informazioni di configurazione sono codificati direttamente mentre sarebbe molto più opportuno leggerli da file esterni garantendo così una maggiore flessibilità dell'intero sistema.

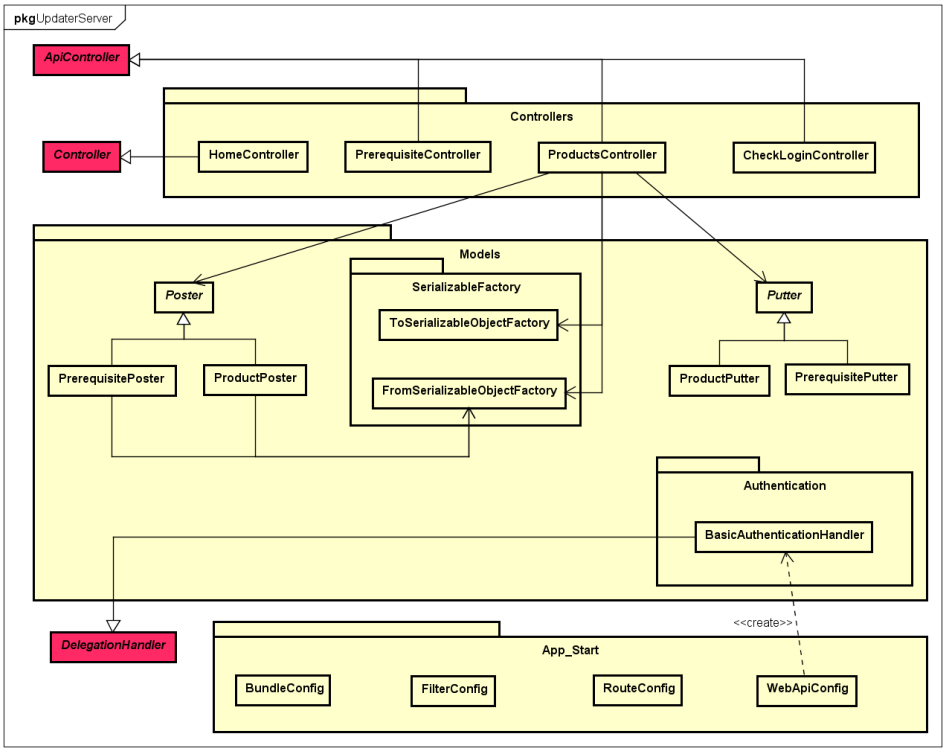

Figura 1.6: Struttura macro componente Updater Server

1.3.6 Updater Database Entity Framework (DBEF)

1.3.6.1 Descrizione

Identifica un'insieme di componenti incaricate di mettere in relazione il paradigma della programmazione orientata agli oggetti e la base di dati relazionale.

1.3.6.2 Responsabilità

- Fornire un accesso facile e semplice alla base di dati relazionale.

1.3.6.3 Implementazione

Dipendenze esterne

- Devart;
- Entity Framework.

Dipendenze interne Nessuna.

Design patter Entity Framework segue il pattern ORM.

L'utilizzo della tecnologia Devart ha permesso di costruire le componenti strutturate su Entity Framework in modo completamente automatico a partire dal modello sql della base di dati.

1.3.6.4 Funzionamento

Le componenti create dalla tecnologia Devart consentono di usare la tecnologia Linq di .NET per poter usufruire attraverso programmazione funzionale di tutto il contenuto del database attraverso un'automatica conversione in oggetti.

1.3.6.5 Criticità e possibili miglioramenti

Le criticità che potrebbero emergere sono principalmente le problematiche del pattern ORM, talora colpevolizzato di non rispettare i principi della programmazione ad oggetti e definito anti-pattern [5]. I vantaggi però del suo uso sono stati evidenti durante lo sviluppo. La tecnologia Devart permetteva l'aggiornamento in pochissimi passi a qualsiasi modifica nel modello della base di dati. L'uso di Entity Framework permette l'uso di Linq e quindi di accedere con una facilità disarmante ai dati nel database attraverso la

[5] http://www.yegor256.com/2014/12/01/orm-offensive-anti-pattern.html http://seldo.com/weblog/2011/08/11/orm_is_an_antipattern.

programmazione funzionale e quindi codificando spesso una singola linea di codice.

1.4 Relazioni esterne tra componenti

1.4.1 Relazioni Updater

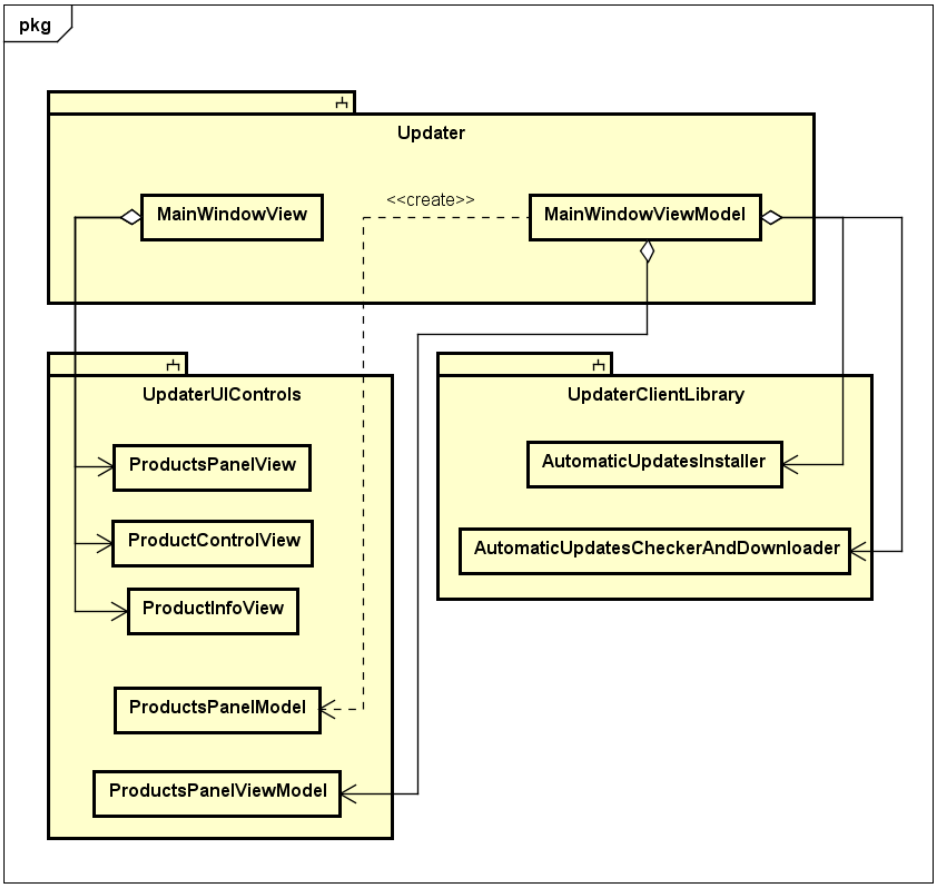

Figura 1.7: Relazioni componenti di Updater con le componenti di UIControls e Client Library

1.4.2 Relazioni Updater UIControls

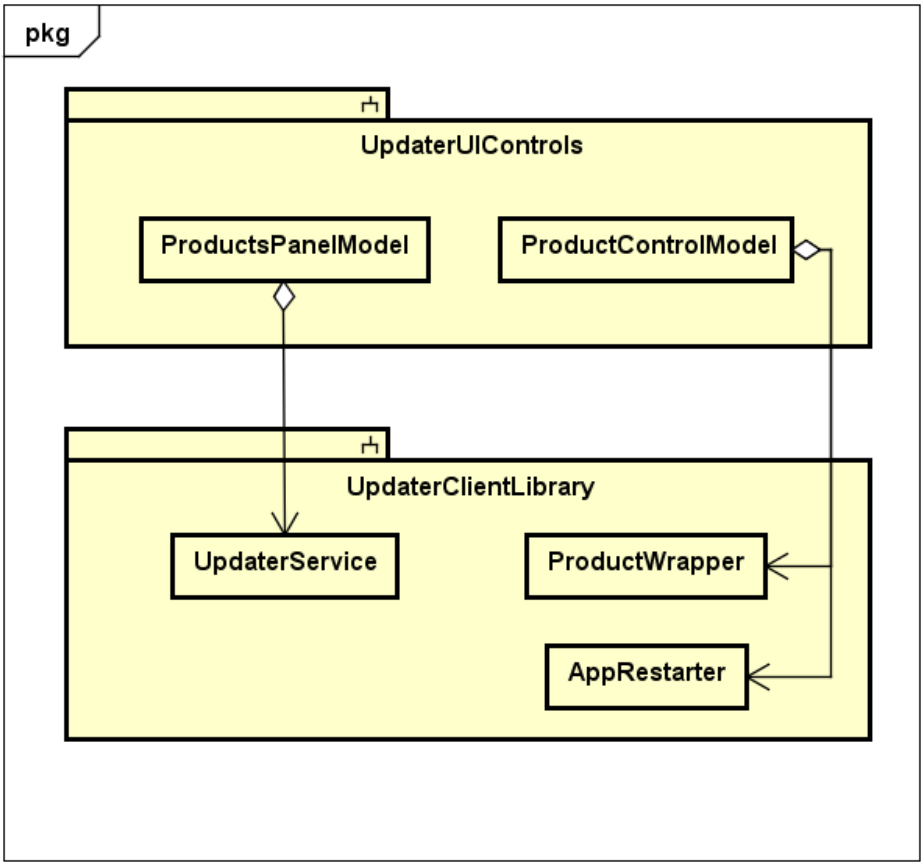

Figura 1.8: Relazioni componenti di Updater UIControls con le componenti di Client Library

1.4.3 Relazioni Updater Client Library

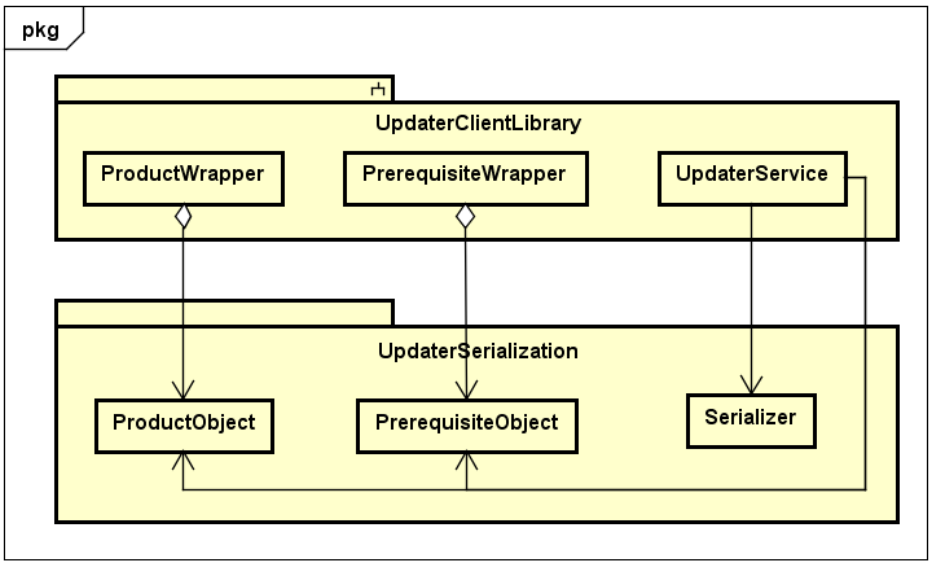

Figura 1.9: Relazioni componenti di Updater Client Library con le componenti di Updater Serialization

1.4.4 Relazioni Updater Server

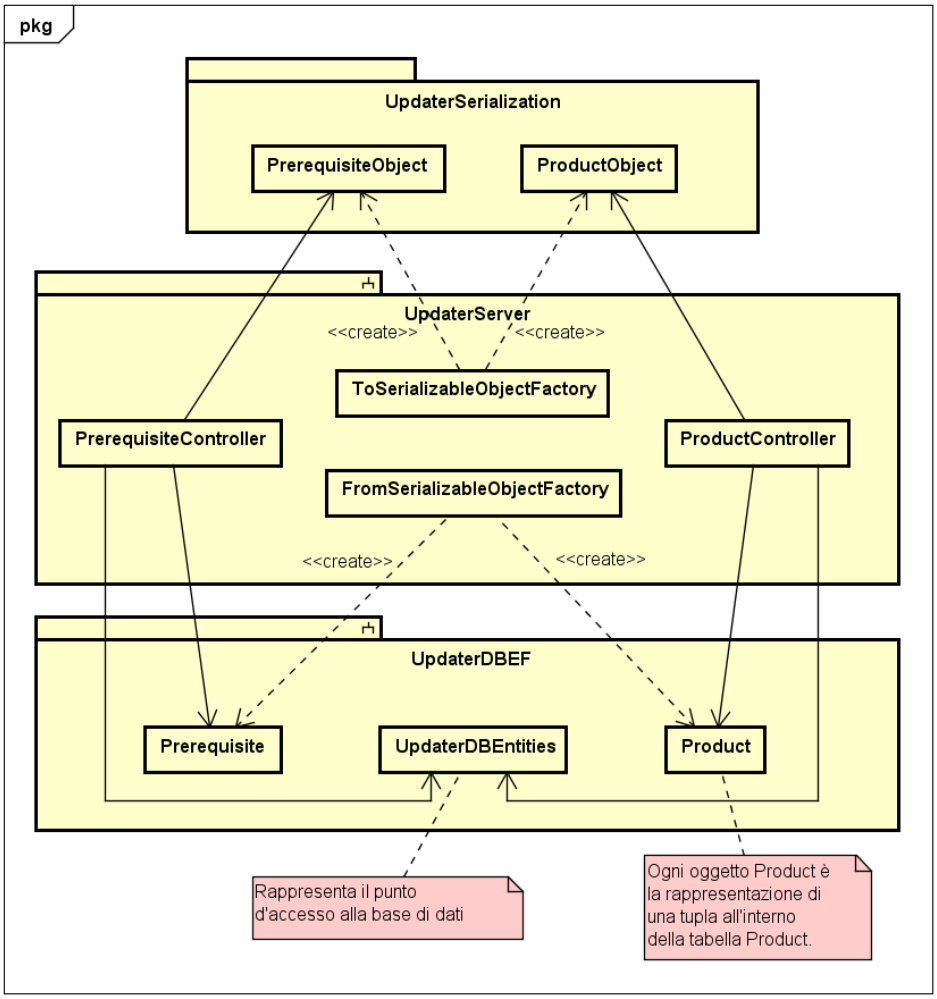

Figura 1.10: Relazioni componenti di Updater Server con le componenti di Updater Serialization e UpdaterEBEF