Case Study:
Designing a Note-Taking App

Learning resource from the course UML and Object-Oriented Design Foundations by Karoly Nyisztor.

Case Study: Designing a Note-Taking App

In this chapter, I walk you through the process of designing a note-taking application.

We'll start by collecting the requirements and writing user stories. We're going to analyze the system from various angles.
By the end of this chapter, we'll have created several UML diagrams that describe the structure and the behavior of our software.

This is going to be an interesting exercise as we put into practice what we learned so far.

Collecting the Requirements

In this lecture, I'm going to illustrate the requirements collection phase through a real example.

Let's start by thinking about the features of a note-taking application. First, we're going to be creating and editing text-based notes. But we don't want to create the next boring note-taking app.

So, let's get creative and add some interesting features.

> How about adding photos to our notes? And capturing hand-drawn sketches would be a cool addition, too.
>
> Privacy has become increasingly important. We should allow the storing of sensitive notes.
>
> Automatic syncing to Dropbox, iCloud or Google Drive? Sure, many users will find that a useful feature.

Now that we have some ideas floating around, we can come up with a first draft of our distilled requirements.

Note-Taking App - Functional Requirements

- We need to build a note-taking app. Users can create an edit text-based notes. A note may also include images or hand-drawn sketches.
- Sensitive notes can be protected from prying eyes using a password.
- The app automatically uploads changes to pre-configured servers.

  We should support all major platforms: Dropbox, iCloud and Google Drive.

Next, let's talk about the nonfunctional requirements.

Note-Taking App - Nonfunctional Requirements

- Let's release our app for iOS first. We'll support iOS 10 and newer versions. The app needs to run on the iPhone and the iPad as well.

- We'll create a dedicated support website and include the link in the app's description and its "About" page.

Now that we have collected the requirements, we can proceed to the next step. We're going to map these requirements to technical descriptions.

Creating User Stories

Now that we've gathered the requirements, we'll be writing user stories. So, let's get started.

These are the functional requirements:

Note-Taking App - Functional Requirements

- We need to build a note-taking app. Users can create an edit text-based notes. A note may also include images or hand-drawn sketches.

- Sensitive notes can be protected from prying eyes using a password.

- The app automatically uploads changes to pre-configured servers.

  We should support all major platforms: Dropbox, iCloud and Google Drive.

Considering these requirements, we identify three major topics:

1. Note creation and editing
2. Privacy - protecting user data
3. Syncing to cloud servers

These are all big chunks of functionality that can't be described through single user stories. Thus, I'm going to create an epic for each. As you may recall, an epic consists of multiple user stories that describe common functionality.

Epic #1: Note creation and editing

- As a user, I want to create and edit notes so that I can quickly jot down my thoughts.

- As a user, I want to attach photos to a note so that I can keep my memories in one place.

- As a user, I want to add handwritten sketches so that I can insert funny drawings into my notes.

Epic #2: Privacy - Protecting User Data

- As a user, I want to create private notes so that only I can access them.

- As a user, I want to protect my sensitive notes with a password.

Epic #3: Syncing to cloud servers

- As a user, I want to sync my notes across my iOS devices so that my data is up-to-date on all of them.

- As a user, I want my notes automatically uploaded to cloud servers (Dropbox, Google Drive or iCloud) so that I have a backup of all my data.

These user stories are technical descriptions that serve as a starting point for our use-case diagrams.
Up next, I'm going to show how we might go ahead and map these user stories to actual use-case diagrams.

Diagraming the Main Use Cases

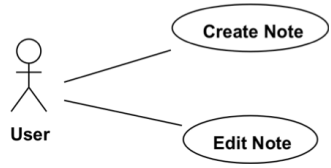
In this lecture, we're going to represent our user stories as use-case diagrams. Let's start with the first epic:

Epic #1: Note creation and editing

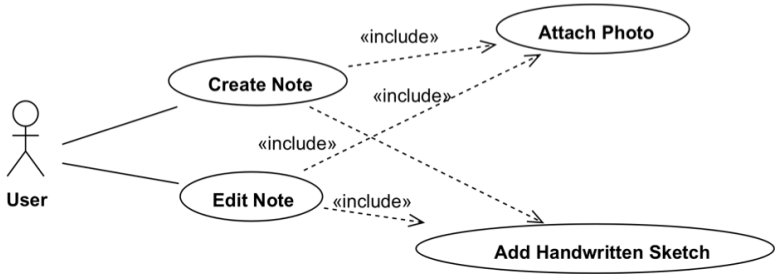
- As a user, I want to create and edit notes so that I can quickly jot down my thoughts.
- As a user, I want to attach photos to a note so that I can keep my memories in one place.
- As a user, I want to add handwritten sketches so that I can insert funny drawings into my notes.

Let's map these user stories to a use-case diagram.

The actor is the user of this app. Next, I add the use-cases: "Create Note" and "Edit Note".



We also need an "Attach Photo" and "Add Handwritten Sketch" use case. Now, these are not standalone use-cases. We can't attach a photo or add a handwritten sketch without creating or editing a note. Thus, I represent them as included in the "Create Note" and "Edit Note" use-cases.



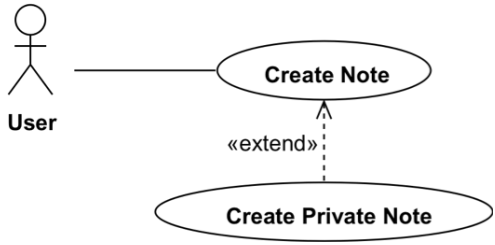
Let's continue with the second epic.

Epic #2: Privacy - Protecting User Data

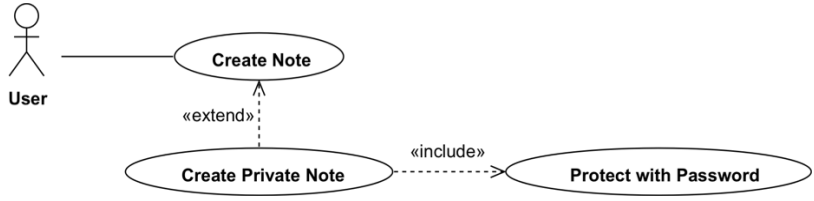
- As a user, I want to create private notes so that only I can access them.
- As a user, I want to protect my sensitive notes with a password.

Again, we need an actor; it's the user as usual.

The creation of private notes is a special case of note creation. We can represent the "Create Private Note" as an extension of the regular "Create Note" use-case.



We need to protect sensitive notes with a password, so I create a "Protect with Password" use case for it. This use-case should be included in the Create Private Note use-case, so I draw it using an include relationship.



Here's the third epic:

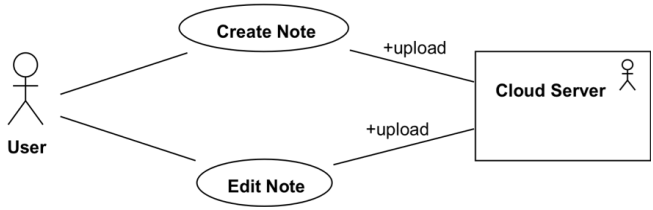
Epic #3: Syncing to cloud servers

- As a user, I want to sync my notes across my iOS devices so that my data is up-to-date on all of them.
- As a user, I want my notes automatically uploaded to cloud

servers (Dropbox, Google Drive or iCloud) so that I have a backup of all my data.

This epic is about synchronizing data with a server. This server is another actor, a non-human one. I'll represent it on the right side of the diagram using this special format.

Let's determine the use-cases.
So, whenever we create or modify a note, it needs to be sync'd with the server.



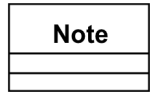
That's an over simplification, but it illustrates what needs to happen in this scenario. As you may recall, use-case diagrams are a means to share our ideas with nontechnical people, so try to keep it simple.

Next, we get into more technical details, as we'll start to identify our classes.

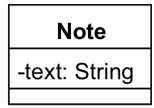
Identifying the Classes

Let's create the static structure of our system. We'll identify the main classes and the relationships between them.

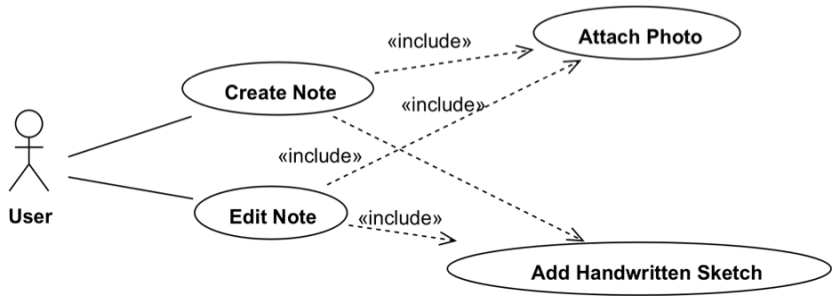
Our app is about taking notes. So, we'll need a class that represents a Note.



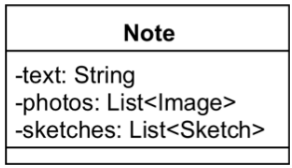
A Note has a text. So, I add a private attribute called "text" of type String. As you may recall, we should not expose class properties. Hence "text" is private.



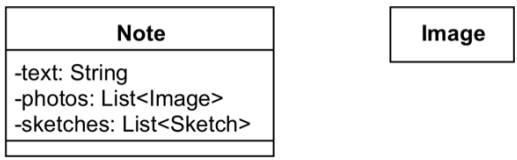
What else do we know? Let's take a look at the use-case diagrams we've put together.



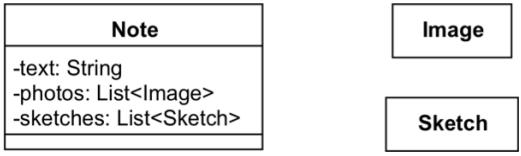
We also need an "Attach Photo" and "Add Handwritten Sketch" use case. A Note needs to have an attribute for the photos and one for the handwritten sketches.
I'm using the plural form: photos and sketches. That is, a Note may have many photos and hand-drawn sketches attached to it. So, I'll make these attributes of List type.



The photos attribute is a List of Image type. I introduce an Image class, which can later be changed to an existing type. In iOS, that would be UIImage, but that's not important. The point is to identify the potential classes that play a role in our system.

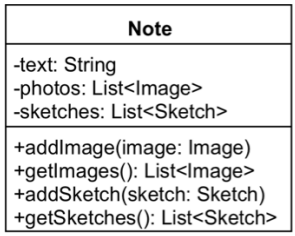


Similarly, we need a list of sketches. The Sketch class represents our hand-drawn sketches.



The Note class needs some methods:

- *addImage(image: Image)* to attach a new image and
- *getImages(): List<Image>* to retrieve all images of a Note.
- *addSketch(sketch: Sketch)* and
- *getSketches(): List<Sketch>* to get the hand-draw sketches of a Note



We don't know anything yet about the underlying format for the Image and the Sketch class.

And that's fine. We need to abstract things first and iteratively refine it. At the end, we may store our hand-drawn sketches either as a JPEG image. But requirements may change and we'll need a vector format, so we'll use PDF.
Again, we should not go into such details at this stage or we may easily get stuck. This phenomenon is well-know and it even has a name: analysis-paralysis.

> *Start with a few broad strokes instead of overthinking and spending too much time on figuring out the details right away. Then try to get more specific as you understand more.*

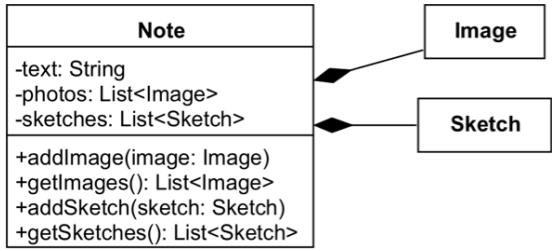
Now, let's think about the relationships between these classes? Is there an association between the Note and the Image class? Or rather a dependency?

The Note class has an attribute that refers to the Image and the Sketch class. It did not receive instances of these classes as parameters to a method, so it's not a dependency relationship.

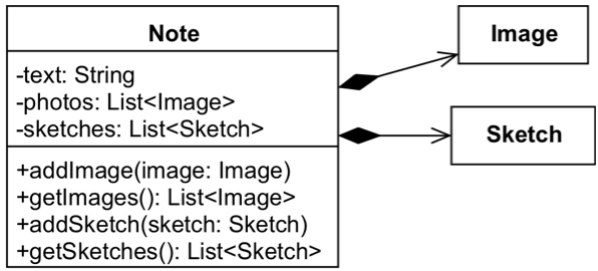
So, is it an association? Yes, it is. But let's dig a bit deeper.

What happens when we delete a Note object with its images and sketches? They will be destroyed, too. It doesn't make sense to keep them if the user decides to remove the Note.

That means that the images and the sketches live and die with the Note object. As you may recall, this is the "part-of" relationship called composition.



Now, an Image doesn't need to know about the Note. Nor does the Sketch. So, these are directed relationships.

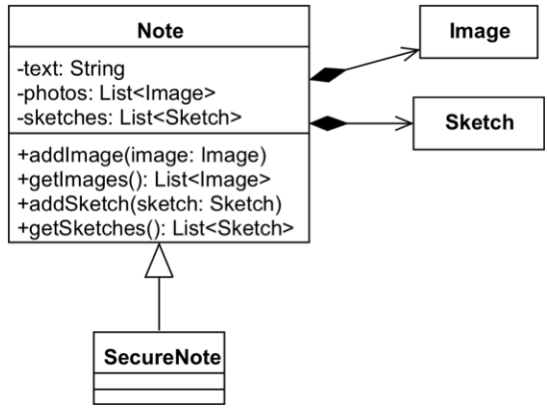


Based on the second epic, we need a specialized Note that holds sensitive data.

Epic #2: Privacy - Protecting User Data

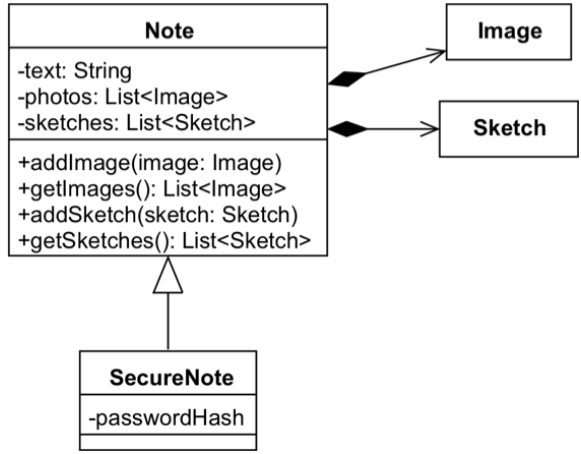
- As a user, I want to create private notes so that only I can access them.
- As a user, I want to protect my sensitive notes with a password.

This note shares most of the attributes and behavior associated with the Note class. This looks like a perfect candidate for inheritance.



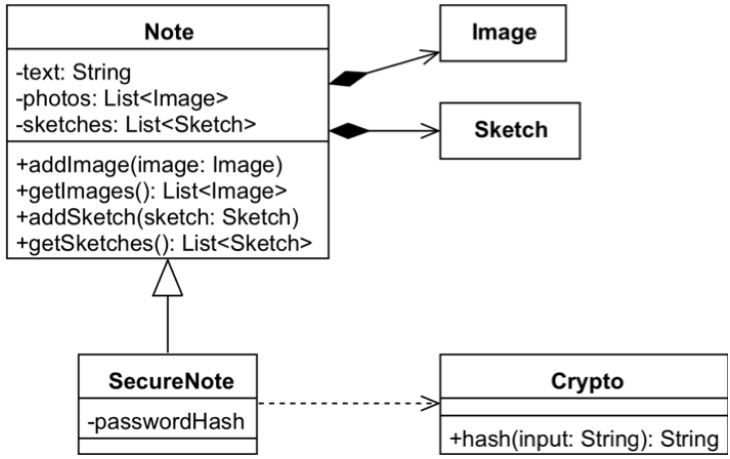
SecureNote inherits from the *Note* class.

In addition to the inherited attributes, it has a property called *passwordHash*.



Storing the password is insecure. Instead, we store the password's hash value. The hash is generated using a one-way hashing algorithm from the password. The password can't be reconstructed from its hash value.

For the hashing algorithm, I'm going to define a *Crypto* class. It provides the public *hash()* method that takes a String as input and returns its hash value. The *SecureNote* is going to rely on the *Crypto* utility class to create the password hash. I indicate this as a dependency between the *SecureNote* and the *Crypto* class.



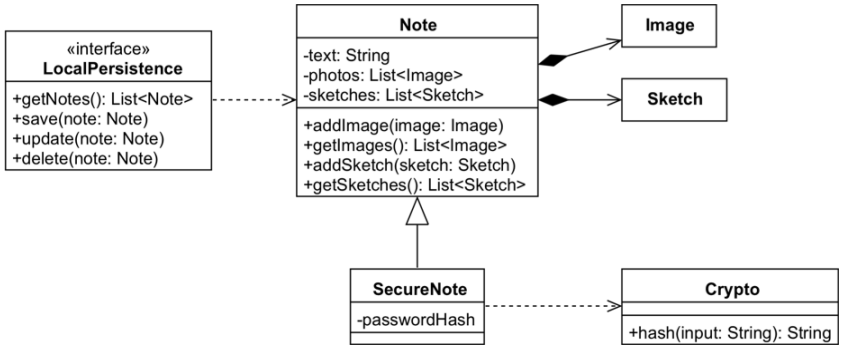
Next, we need a class that's responsible for storing the notes and their associated data in the local persistence. We don't want to be too specific at this point as we haven't defined yet what local persistence means. It could be the filesystem or an SQLite database. We could also choose CoreData.

That's not important at this point, so we'll use abstraction. Instead of specifying a concrete file or database manager, I'm going to create an interface that defines a couple of methods.

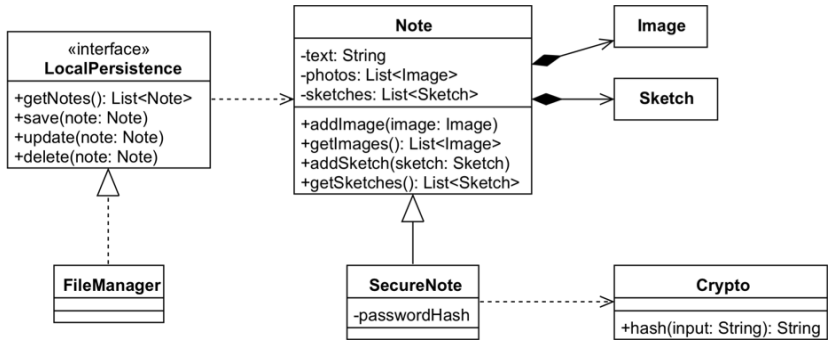
Let's call it *LocalPersistence*. It's an interface: it declares the method signatures that need to be implemented, but it provides no functionality. The implementation classes will be responsible for implementing these methods. The *LocalPersistence* declares the following interface:

- *getNotes(): List<Node>* - retrieves all notes from the local persistence
- *save(note: Note)* - stores a note locally
- *update(note: Note)* - updates a note that has been persisted
- *delete(note: Note)* - removes a note from the local persistence

All these methods have parameters or return values of type *Note*. Thus, we can draw a dependency relationship between the *LocalPersistence* interface and the *Note* class.



Let's say that we decide to store our notes in the file system. The *FileManager* implements the methods declared in *LocalPersistence*. I use the realization relationship to show that.



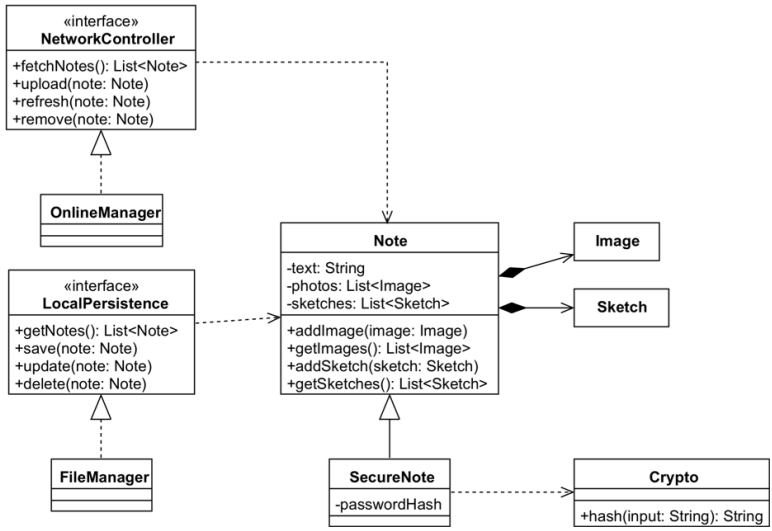
Similarly, I'll create an interface for the cloud syncing feature.
The NetworkController interface declares the methods that take care of the networking-related operations*:

- *fetchNotes(): List<Node>* - fetches all notes from the server
- *upload(note: Note)* - uploads a new note to the server
- *refresh(note: Note)* - updates the note that has been uploaded
- *remove(note: Note)* - deletes a note from the server

Networking is slow, so these should be implemented as asynchronous operations.

(*)Network controllers are more complex but let's keep it simple for this

example.



By now, you've probably got an idea of how class diagrams are created. Now that we mapped the static structure of our system, we can start analyzing its behavior.

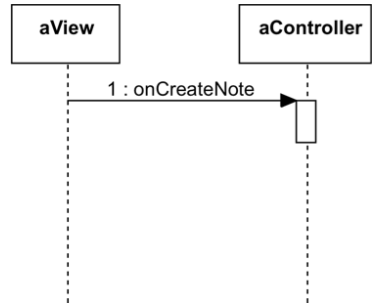
Describing the Flow of Note Creation Using Sequence Diagrams

I'm going to walk you through the creation of the sequence diagram for one specific scenario: adding a new note.

We'll be focusing on the flow of note creation. We try to answer the following question:

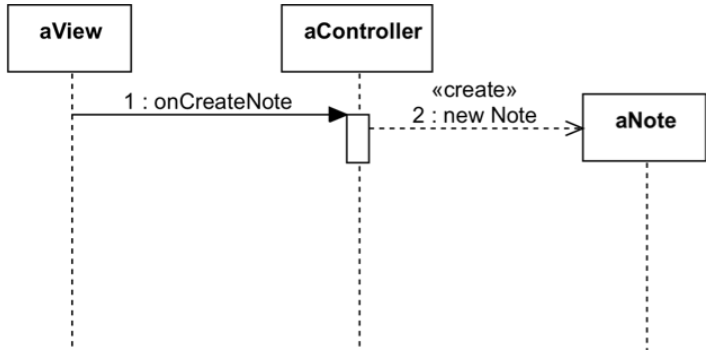
> *Which objects are involved and what messages are sent between them?*

The user can initiate the note creation by pressing a button on the application's user interface. So, we need a view instance first that represents the button. The button receives the user input and triggers and event that's intercepted by a controller object.



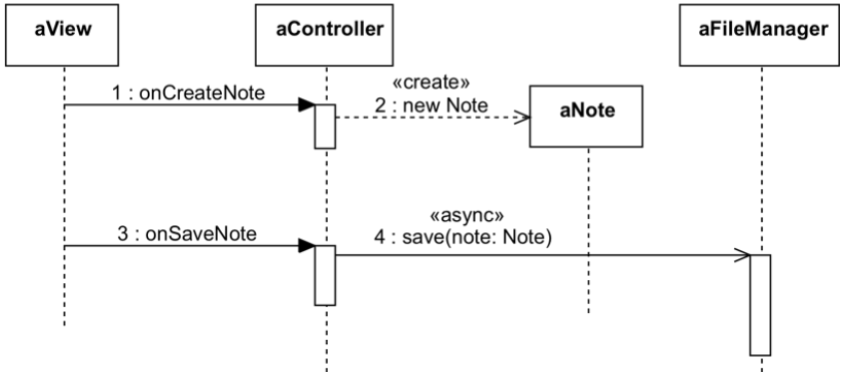
Having separate view and controller objects stands at the core of a well-known architectural design pattern called *Model-View-Controller*. The view displays data and receives user input. The controller triggers actions based on events received from its view and processes the data to be displayed.

In our scenario, the controller triggers the creation of a new *Note* instance. We'll use a create message rather than a regular one. A *Note* object gets created.

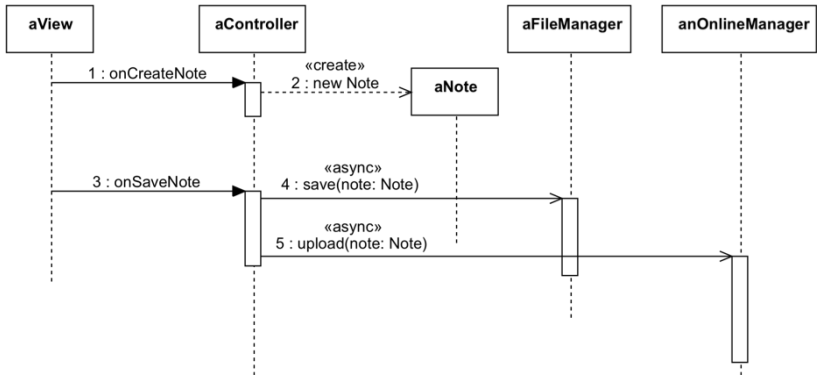


Next, the user fills the note's details and presses the save button. This will trigger two actions: saving to the local persistence and uploading the new note to the cloud.

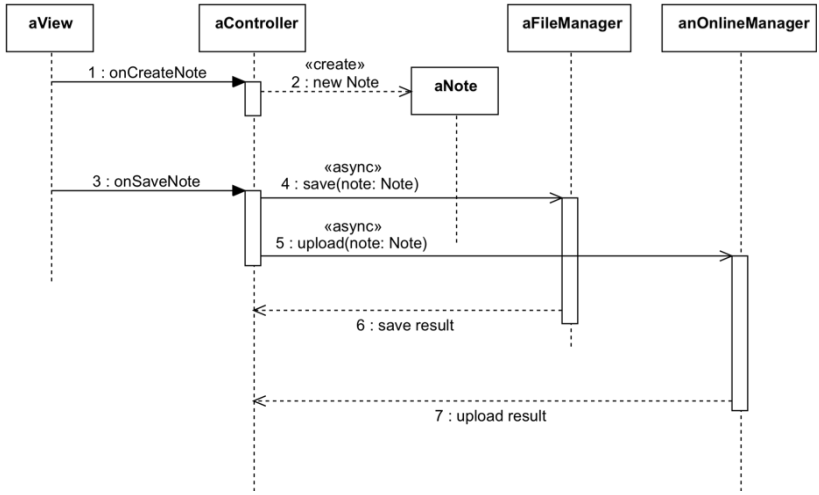
Local persistence is managed by the *FileManager* object. I invoke the *save(note: Note)* method. File operations are slow, thus I call the save method asynchronously. To avoid misunderstanding, I mark the message explicitly as async.



For the upload part, we need an *OnlineManager* instance. The *upload(note: Note)* method gets executed in the background, too.



The saving to local persistence and uploading the new Note to the cloud are asynchronous, so they return instantly, without blocking the caller. Eventually, they return to signal either success or failure.



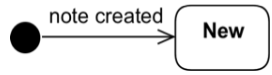
This sequence diagram tells us a lot about the objects and how they interact in the note creation scenario.
We can provide more details by adding further objects and messages as needed.

The Note Object's Statechart Diagram

Let's take a closer look at the possible states of a note object. A note object is created, saved and that's it, right? Well, not quite.

Let's start analyzing the possible states and what could go wrong. We'll soon realize that we need to represent more states than we originally assumed.

The Note object's statechart diagram starts with an initial state. When we create a note, it's in the *New* state. This condition is triggered by the creation event.

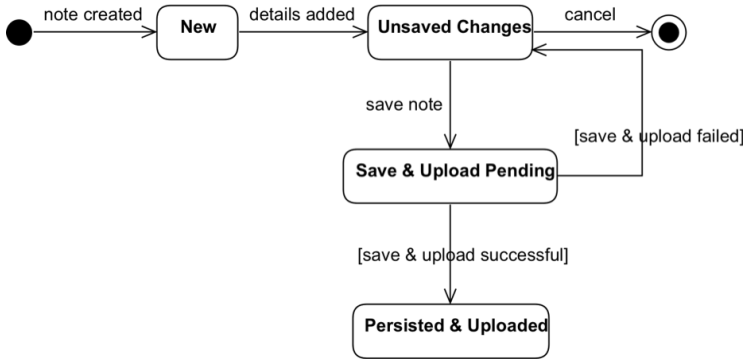


After a new note is created, the user needs to fill in some details. At the end of this process, the note will have unsaved changes. We need a state to express this condition.

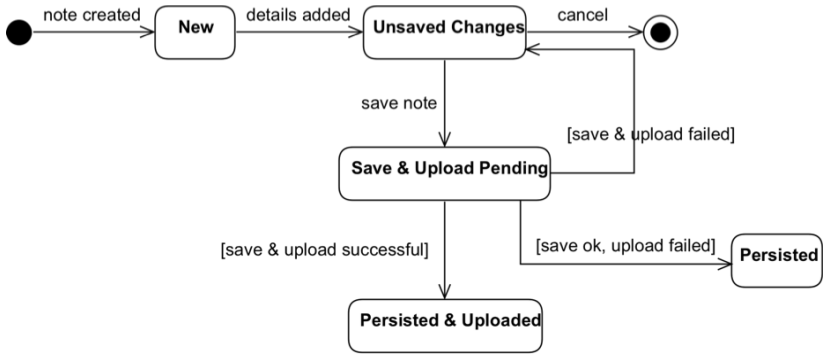
Now, the user can save the Note. He may also decide to cancel the process, which means that our state machine reached its final state.

Saving a note implies storing it in the local persistence and uploading it to the cloud. These are two actions that could potentially fail.

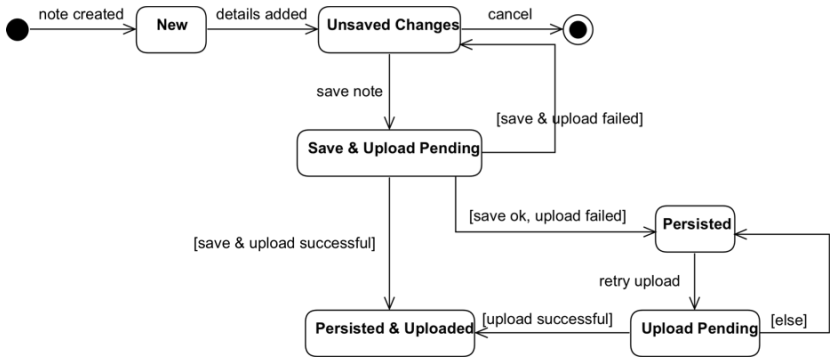
If both fail, the state of our Note object turns back to *Unsaved Changes*. Otherwise, we switch to the *Persisted & Uploaded* state.



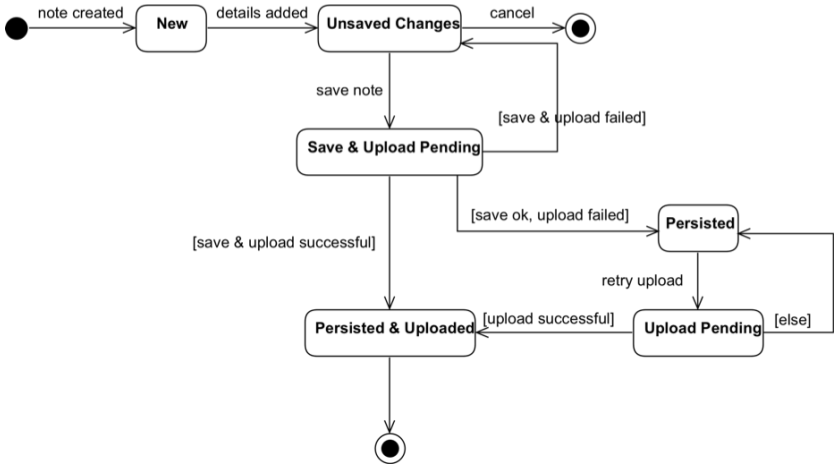
Storing a new note in the file system will usually complete without issues. However, uploading data to a server could fail for various reasons. The *Persisted* state shows that only the local storage was successful.



The user can retry the uploading action, which changes the state of the note object to *Upload Pending*. If this attempt also fails, we go back to the *Persisted* state. Otherwise, the object's state switches to *Persisted & Uploaded*.

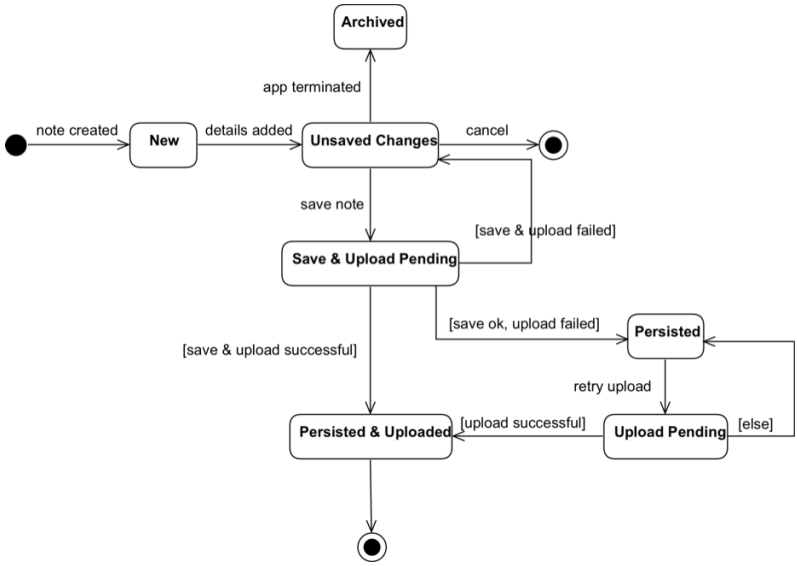


Our note object was successfully created, saved to the local persistence and uploaded to a cloud server. That's the last state so we can complete our state machine by transitioning to the final state.



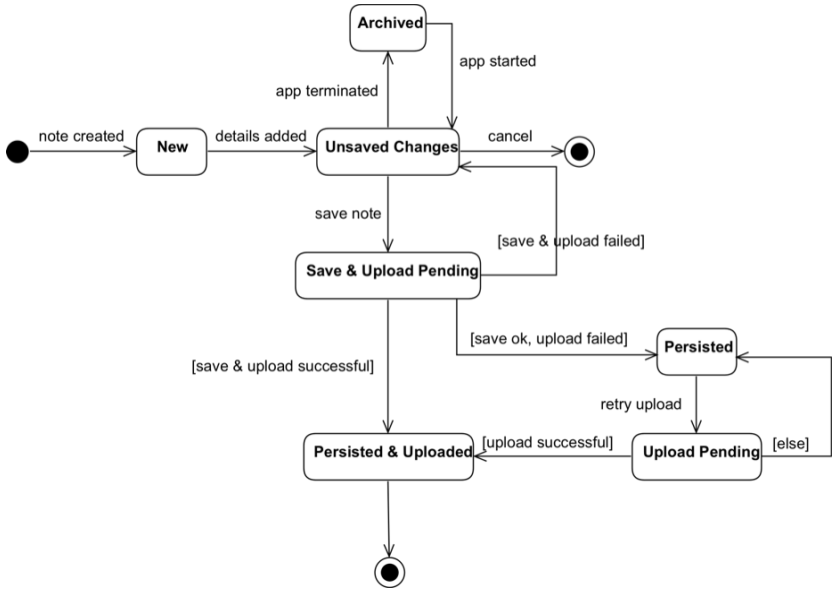
When creating statechart diagrams, it's important to always have a way to exit a state after it's entered. The only exceptions are the initial and the final states.

Let's say that I want to express the *Archived* state. The note object should switch to that state if the app is terminated while the note is in the *Unsaved Changes* state.



When the user starts the app next time, the note remains stuck in this *Archived* state. There's no transition to any other state. This situation is called **deadlock** and it's one of the biggest issues you can encounter with state machines.

To solve the problem, I add a transition to the *Unsaved Changes*. This transition is triggered by the *App Started* event.



So, this is the statechart diagram for the note object. We definitely got more states than we originally assumed.