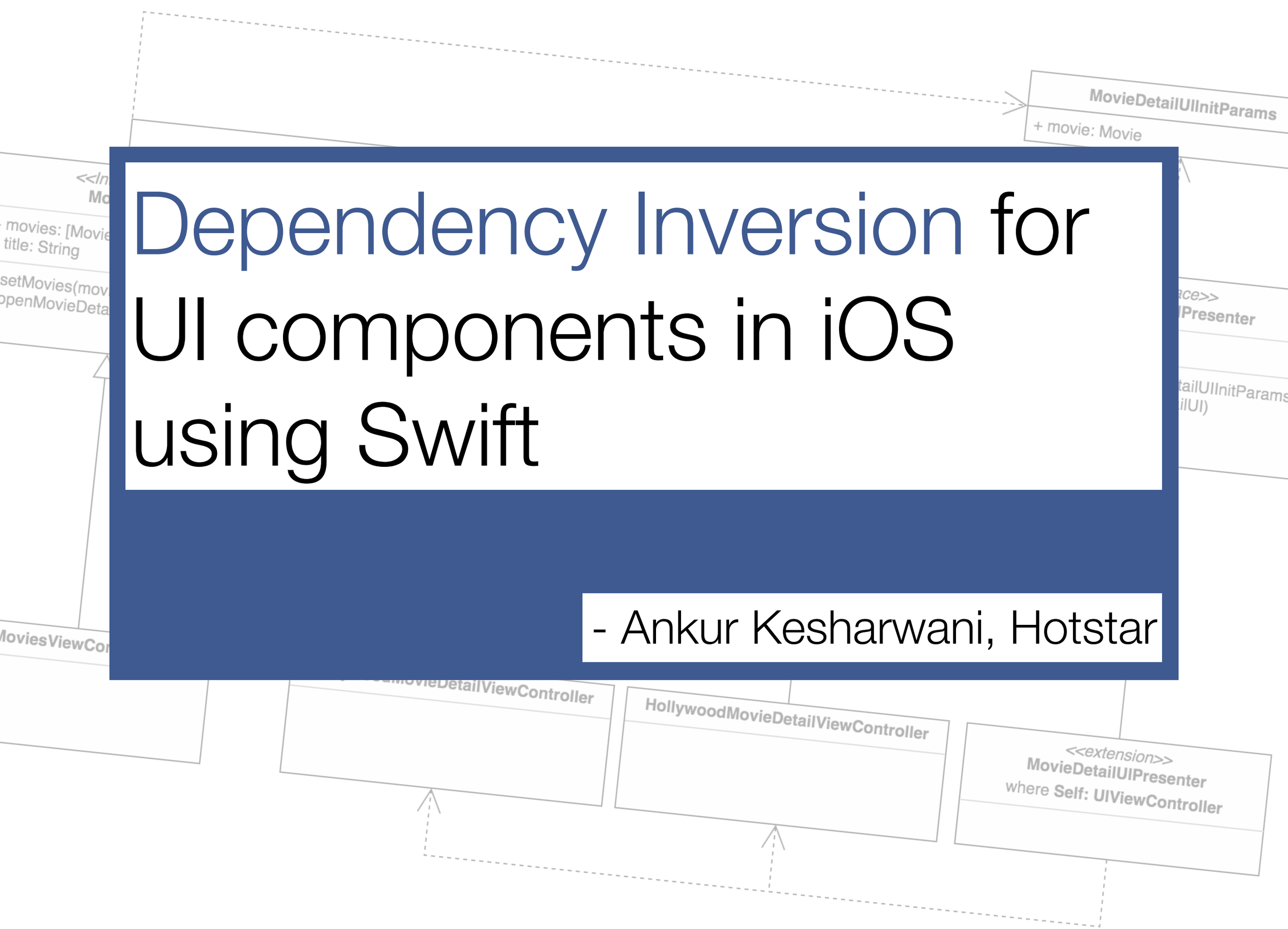
Dependency Inversion for UI components in iOS using Swift

- Ankur Kesharwani, Hotstar



About Me

- 4+ Years of development experience.
- Currently
    - Software Engineer at Hotstar.
- Former
    - Software Engineer, RoundGlass.
    - Founding Member & Tech Lead, Dogether.
    - Team Lead, Frankly
    - Software Engineer, Optimus Information

Understanding Dependency

Understanding Dependency

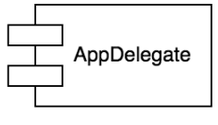
- Dependency is a form of coupling.

Understanding Dependency

- Dependency is a form of coupling.
- In a typical S/W, dependency without any inversion, is from top level modules to lower level modules.

Understanding Dependency

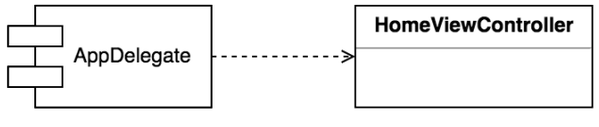
- Dependency is a form of coupling.
- In a typical S/W, dependency without any inversion, is from top level modules to lower level modules.



Understanding Dependency

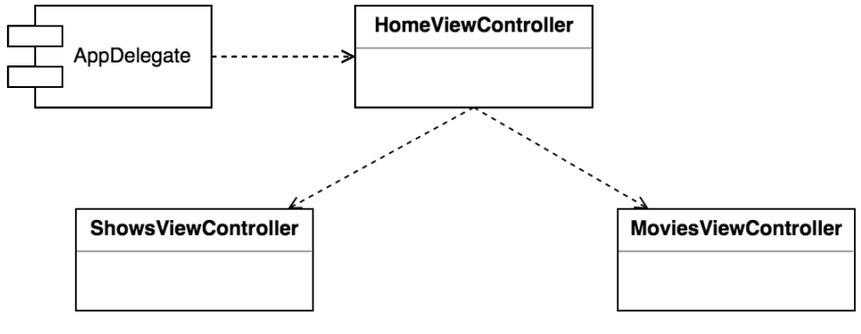
- Dependency is a form of coupling.

- In a typical S/W, dependency without any inversion, is from top level modules to lower level modules.



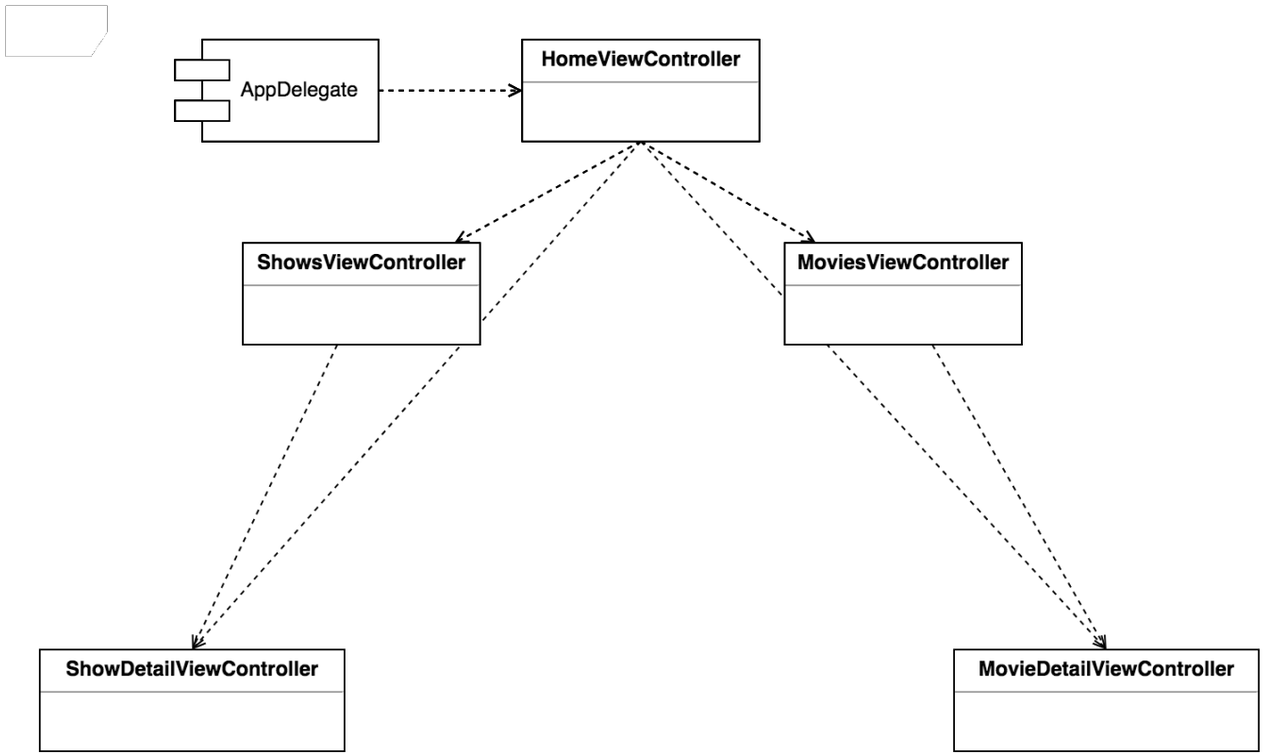
Understanding Dependency

- Dependency is a form of coupling.
- In a typical S/W, dependency without any inversion, is from top level modules to lower level modules.



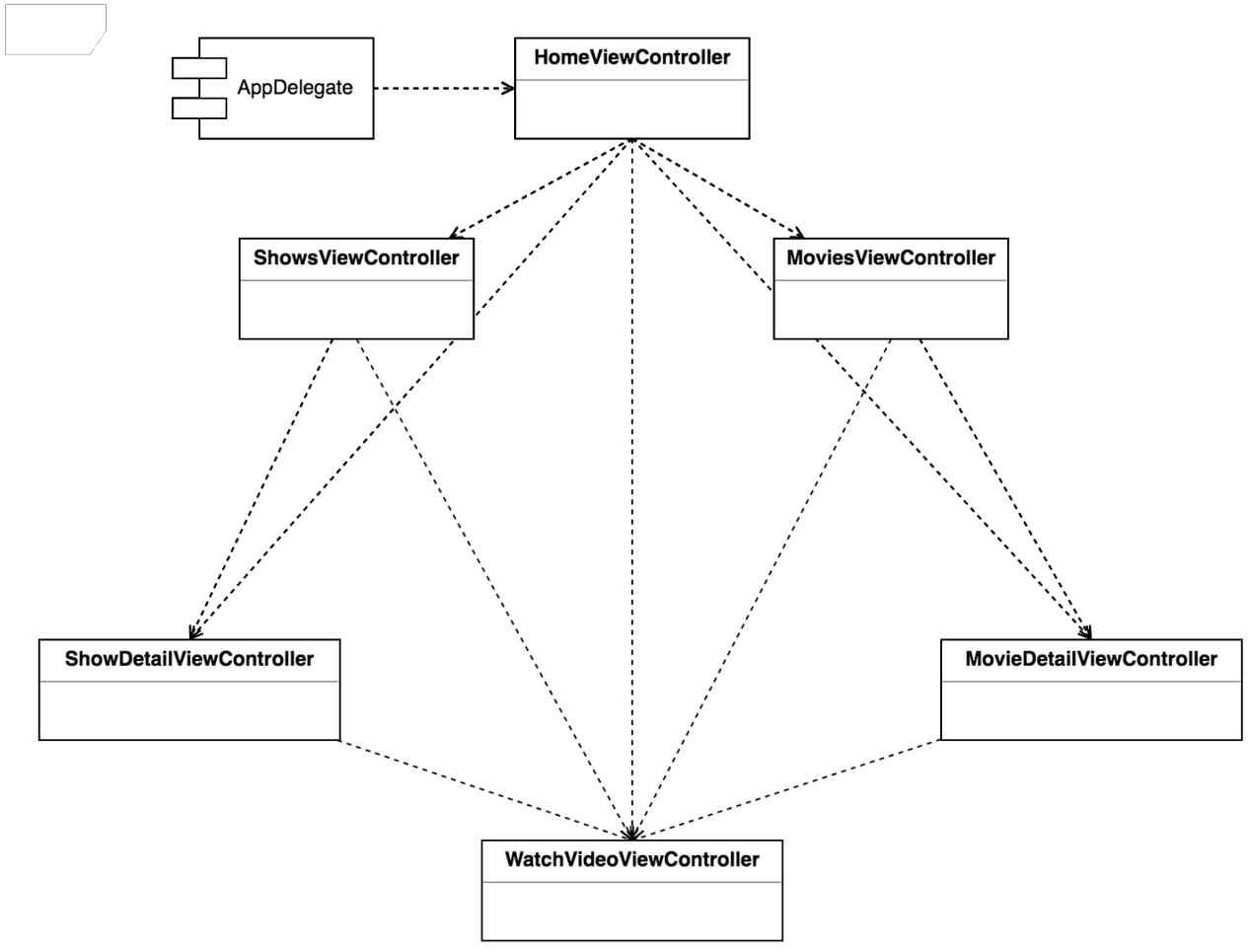
Understanding Dependency

- Dependency is a form of coupling.
- In a typical S/W, dependency without any inversion, is from top level modules to lower level modules.



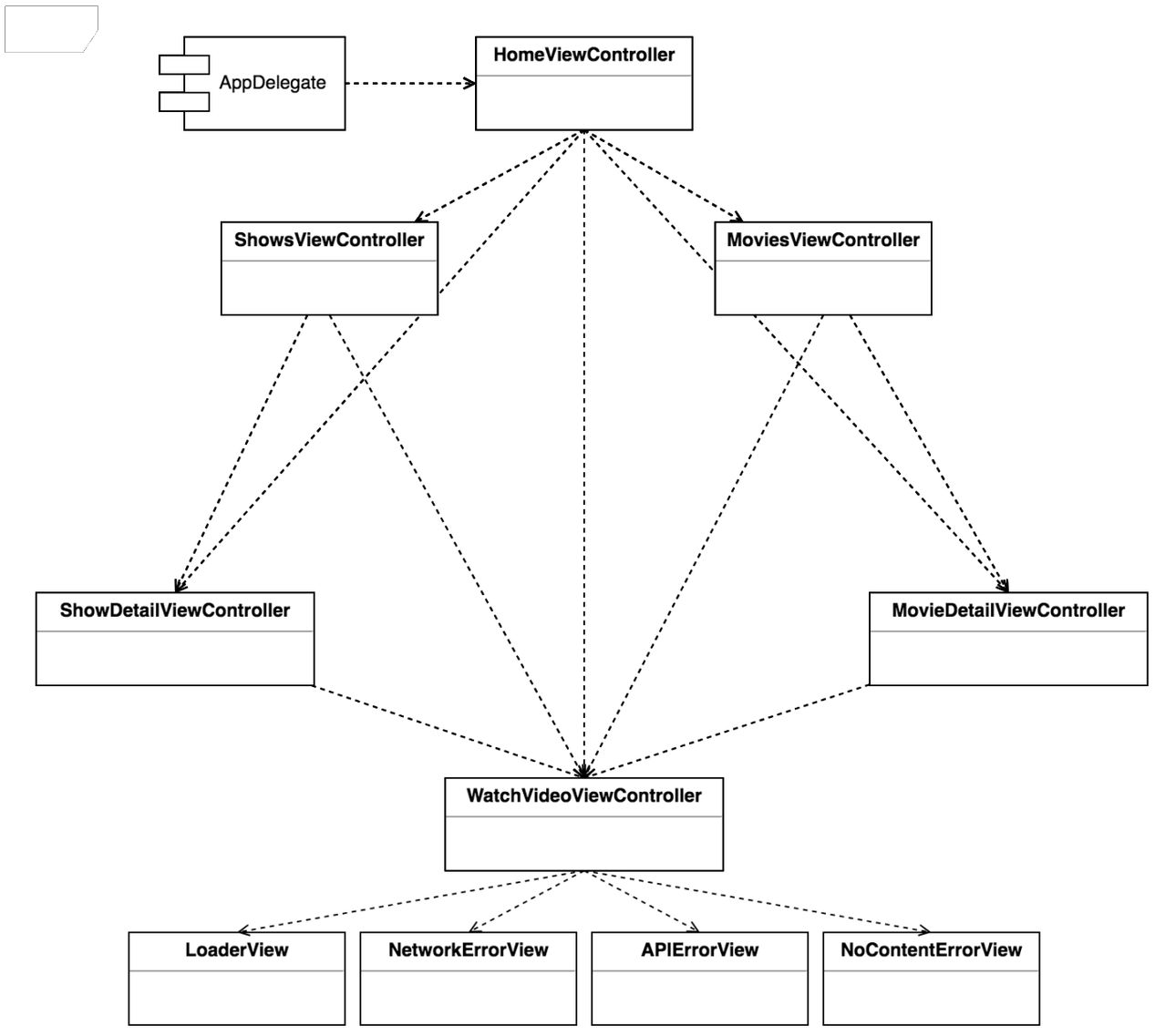
Understanding Dependency

- Dependency is a form of coupling.
- In a typical S/W, dependency without any inversion, is from top level modules to lower level modules.



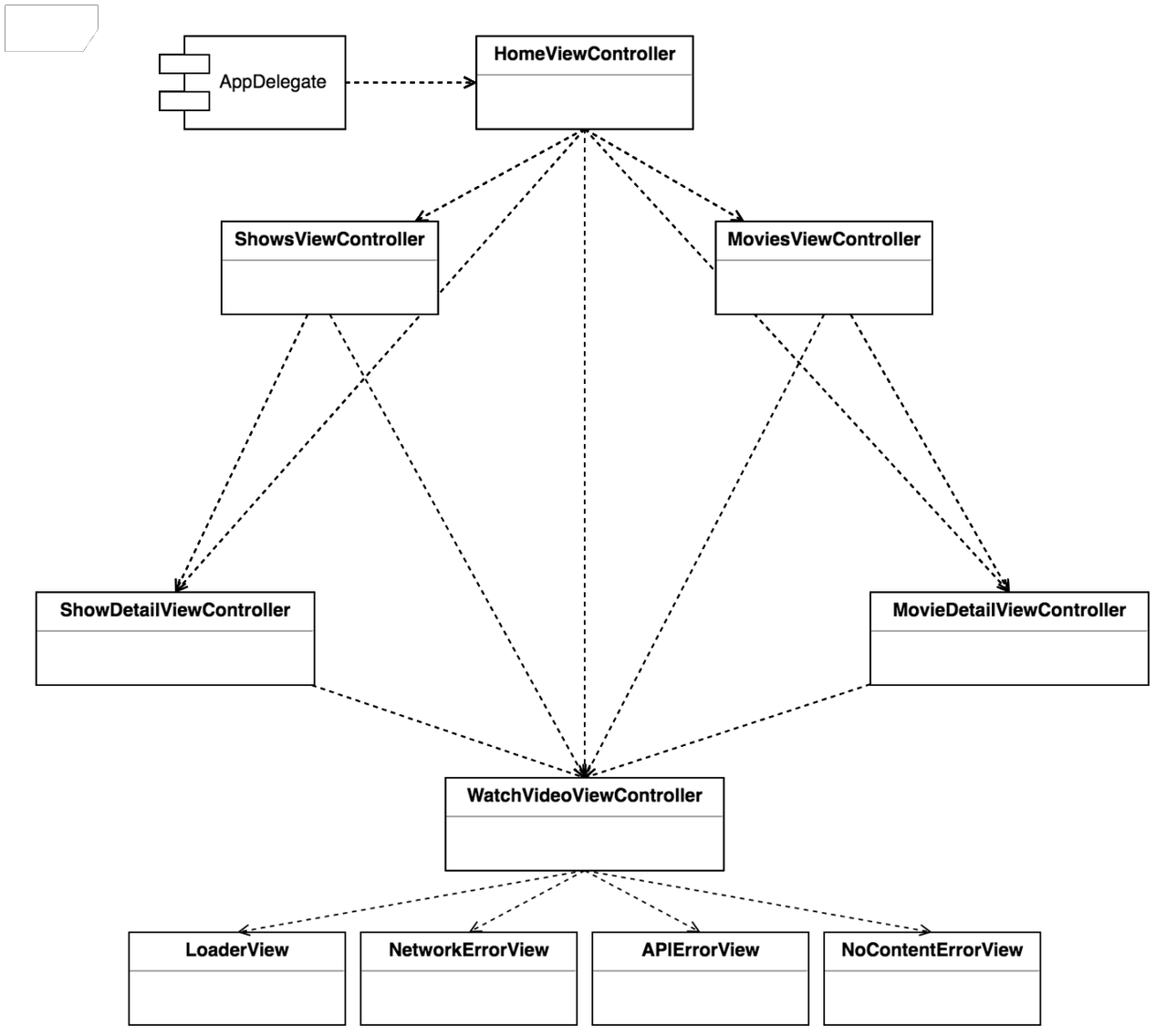
Understanding Dependency

- Dependency is a form of coupling.
- In a typical S/W, dependency without any inversion, is from top level modules to lower level modules.



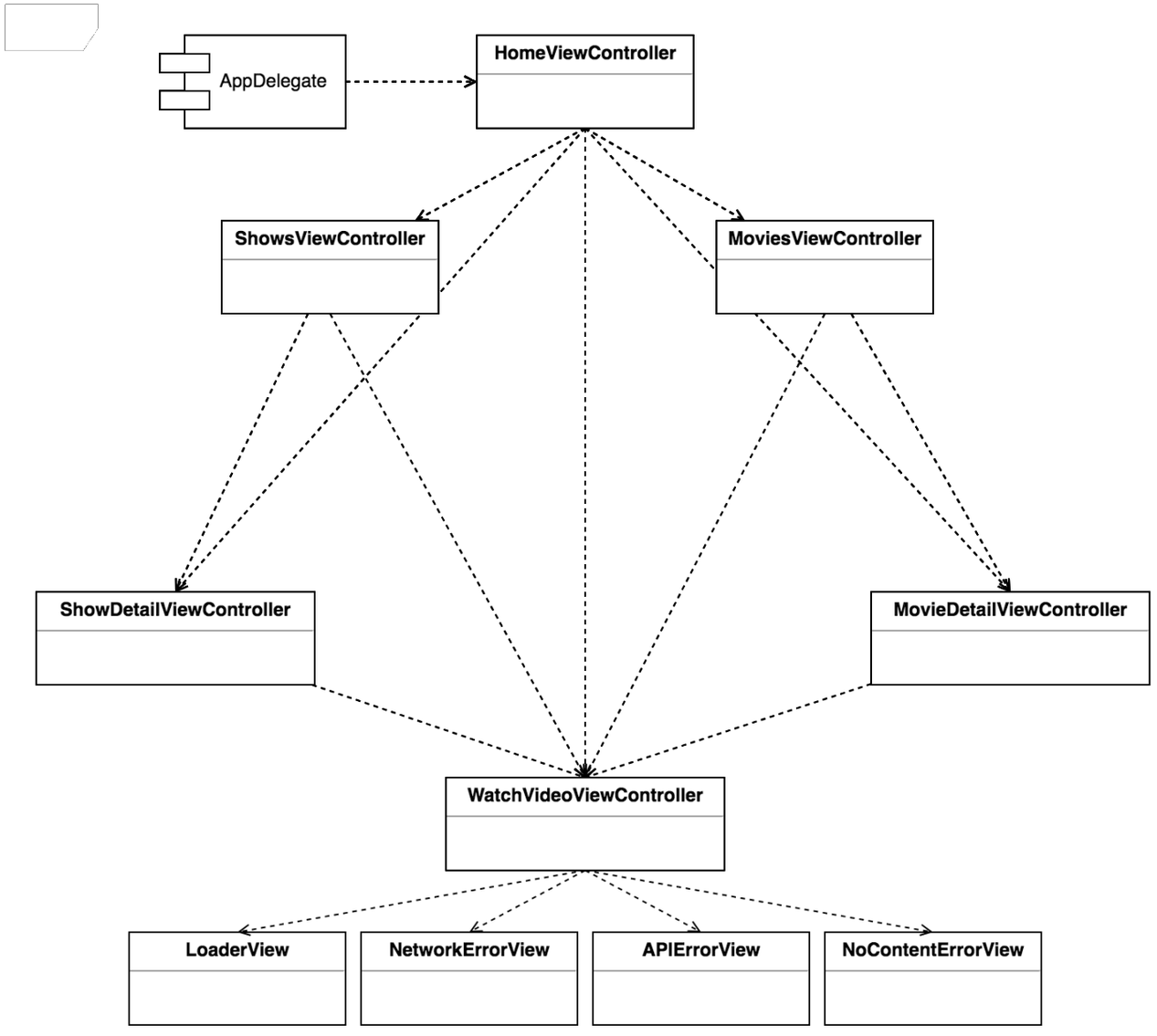
Understanding Dependency

- Dependency is a form of coupling.
- In a typical S/W, dependency without any inversion, is from top level modules to lower level modules.
- This dependency flow makes top level modules dependent on lower level modules.



Understanding Dependency

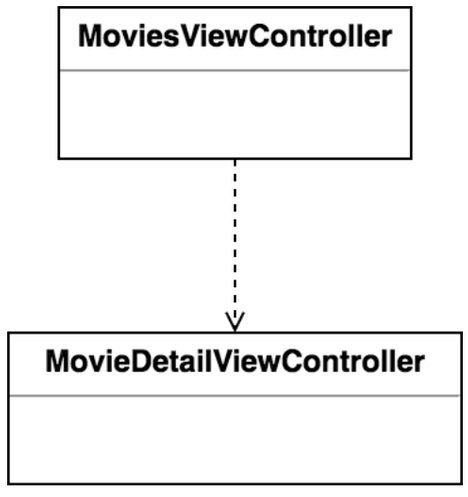
- Dependency is a form of coupling.
- In a typical S/W, dependency without any inversion, is from top level modules to lower level modules.
- This dependency flow makes top level modules dependent on lower level modules.
- When a lower level module changes, it can potentially break a top level module.



Dependency Ex. 1

```
class MoviesViewController: UIViewController {

    private func showMoviesDetailViewController(for movie: Movie) {

        // Instantiate
        let viewController = MoviesDetailViewController.init(nibName: "MoviesDetailViewController",
                                                              bundle: nil)

        // Set variables
        viewController.movie = movie

        // Present
        navigationController?.pushViewController(viewController, animated: true)

    }

}

class MoviesDetailViewController: UIViewController {

    var movie: Movie?

}
```

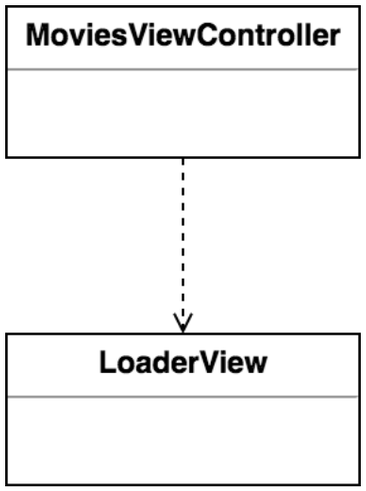
Dependency Ex. 1



Dependency Ex. 2

```
class LoaderView: UIView {

}

class MoviesViewController: UIViewController {

    var loaderView: LoaderView?

    private func showLoader() {
        guard loaderView == nil else {
            return
        }

        let xib = UINib(nibName: "LoaderView", bundle: nil)
        self.loaderView = xib.instantiate(withOwner: nil, options: nil)[0] as? LoaderView

        if let loaderView = self.loaderView {
            view.addSubview(loaderView)
        }
    }

    private func hideLoader() {
        guard loaderView != nil else {
            return
        }

        loaderView?.removeFromSuperview()
    }
}
```

Dependency Ex. 2



The D in SOLID

The D in SOLID

Dependency Inversion principle refers to a specific form of decoupling software modules.

The D in SOLID

Dependency Inversion principle refers to a specific form of decoupling software modules.

It states:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

The D in SOLID

Dependency Inversion principle refers to a specific form of decoupling software modules.

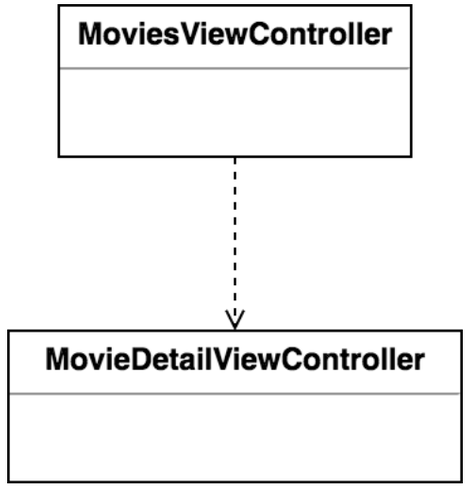
It states:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

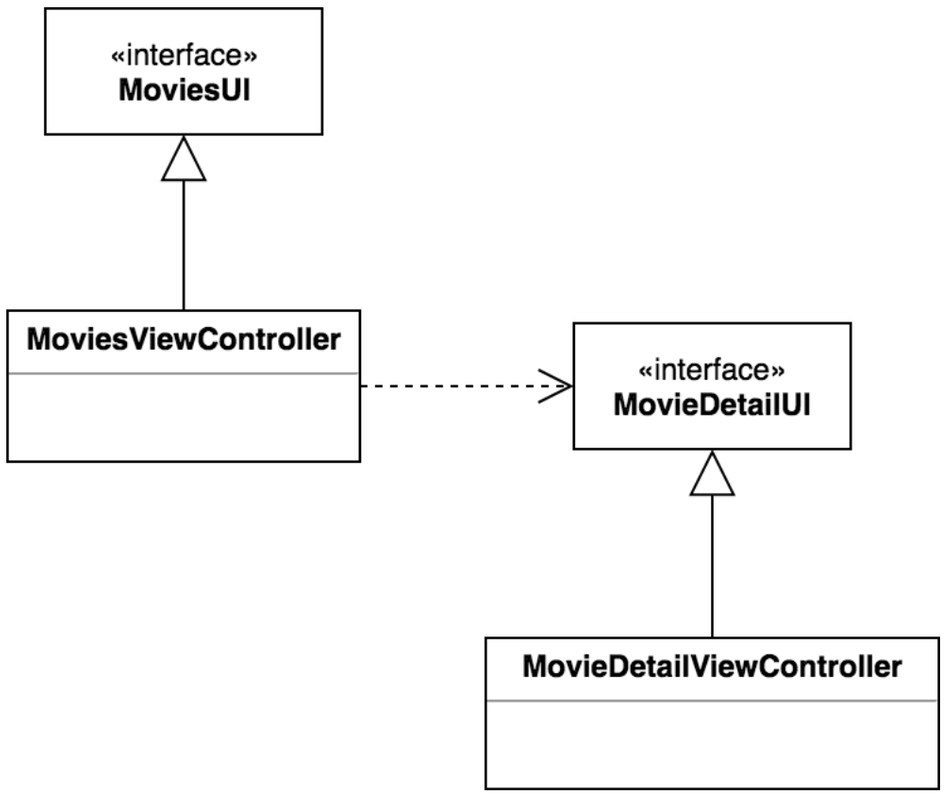
- Abstractions should not depend on details. Details should depend on abstractions.

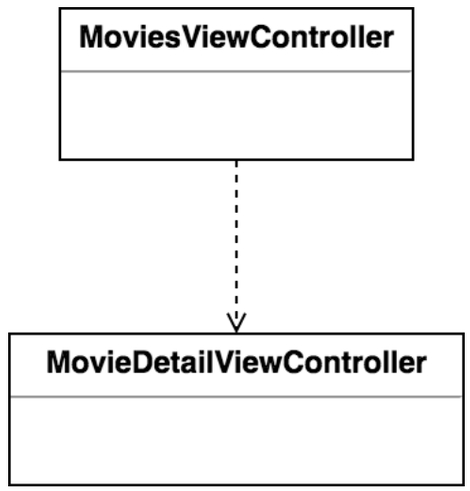
The D in SOLID

The D in SOLID

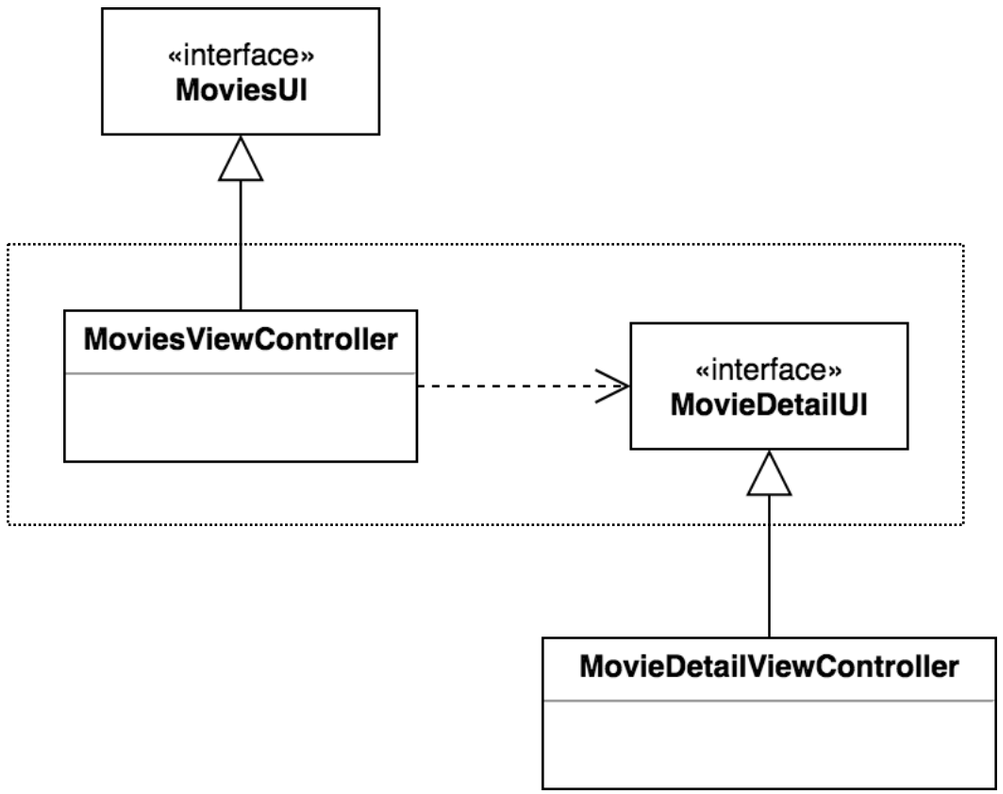


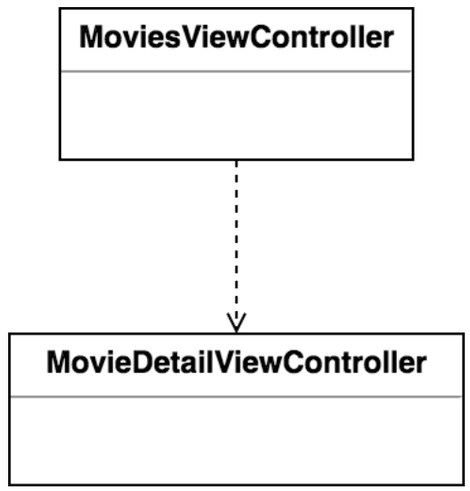
The D in SOLID



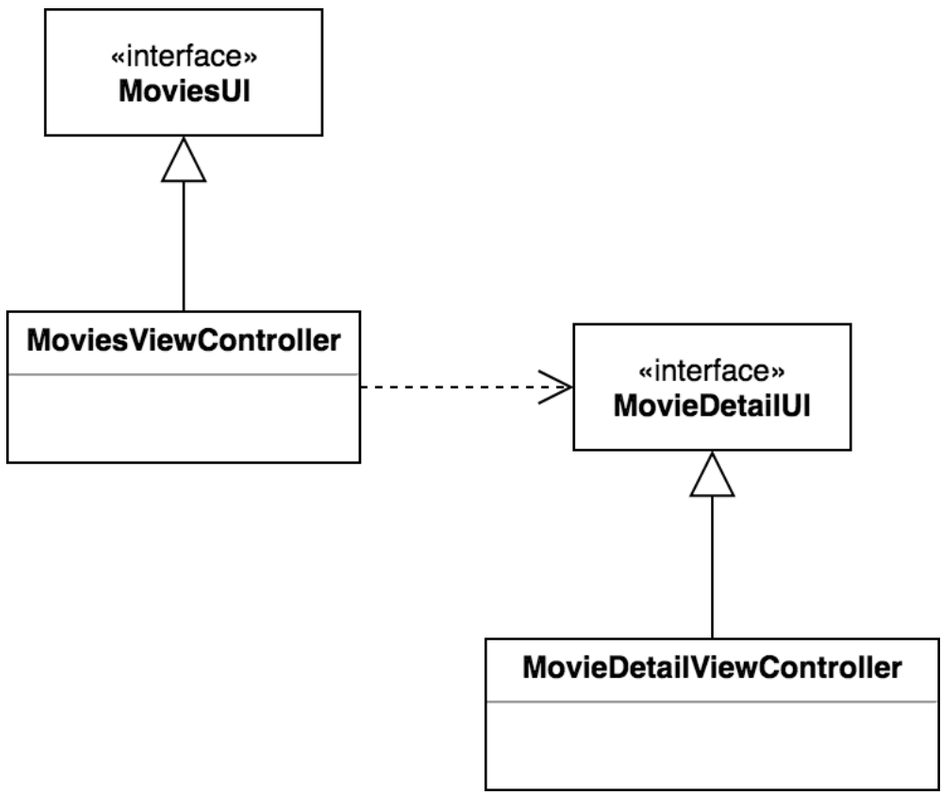


The D in SOLID



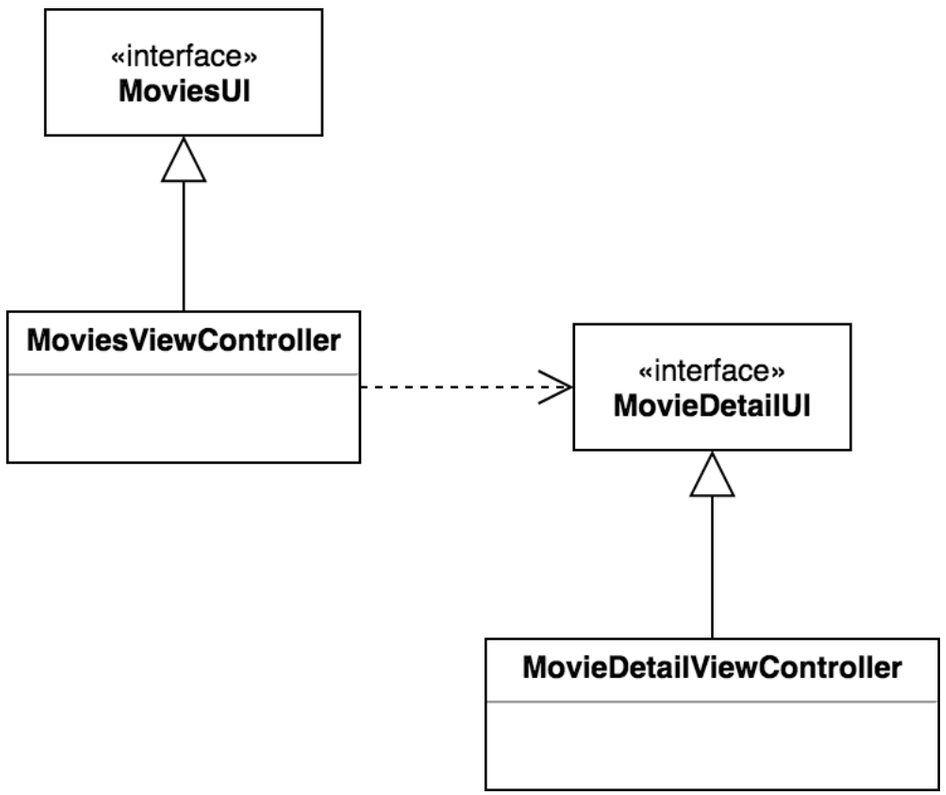


The D in SOLID



- High-level modules should not depend on low-level modules. Both should depend on abstractions.

The D in SOLID



- High-level modules should not depend on low-level modules. Both should depend on abstractions.

- Abstractions should not depend on details. Details should depend on abstractions.

But what about object instantiation?

But what about object instantiation?

Since implementations cannot directly refer other implementations, object instantiation becomes a problem when dependency is inverted.

But what about object instantiation?

Since implementations cannot directly refer other implementations, object instantiation becomes a problem when dependency is inverted.

Solution:

But what about object instantiation?

Since implementations cannot directly refer other implementations, object instantiation becomes a problem when dependency is inverted.

Solution:

- Use Dependency Injection.

But what about object instantiation?

Since implementations cannot directly refer other implementations, object instantiation becomes a problem when dependency is inverted.

Solution:

- Use Dependency Injection.

- Dependency Injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself.

Using Swift Extensions to provide Dependency Injection

Using Swift Extensions to provide Dependency Injection

- In Swift, extensions allow us to extend any Type including Protocols.

Using Swift Extensions to provide Dependency Injection

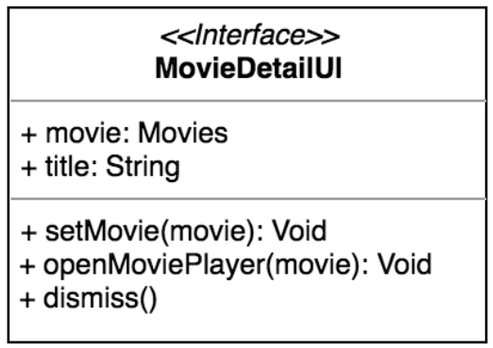
- In Swift, extensions allow us to extend any Type including Protocols.
- When a Protocol is extended using extensions, the implementing class of the Protocol gets all the extended functionalities too.

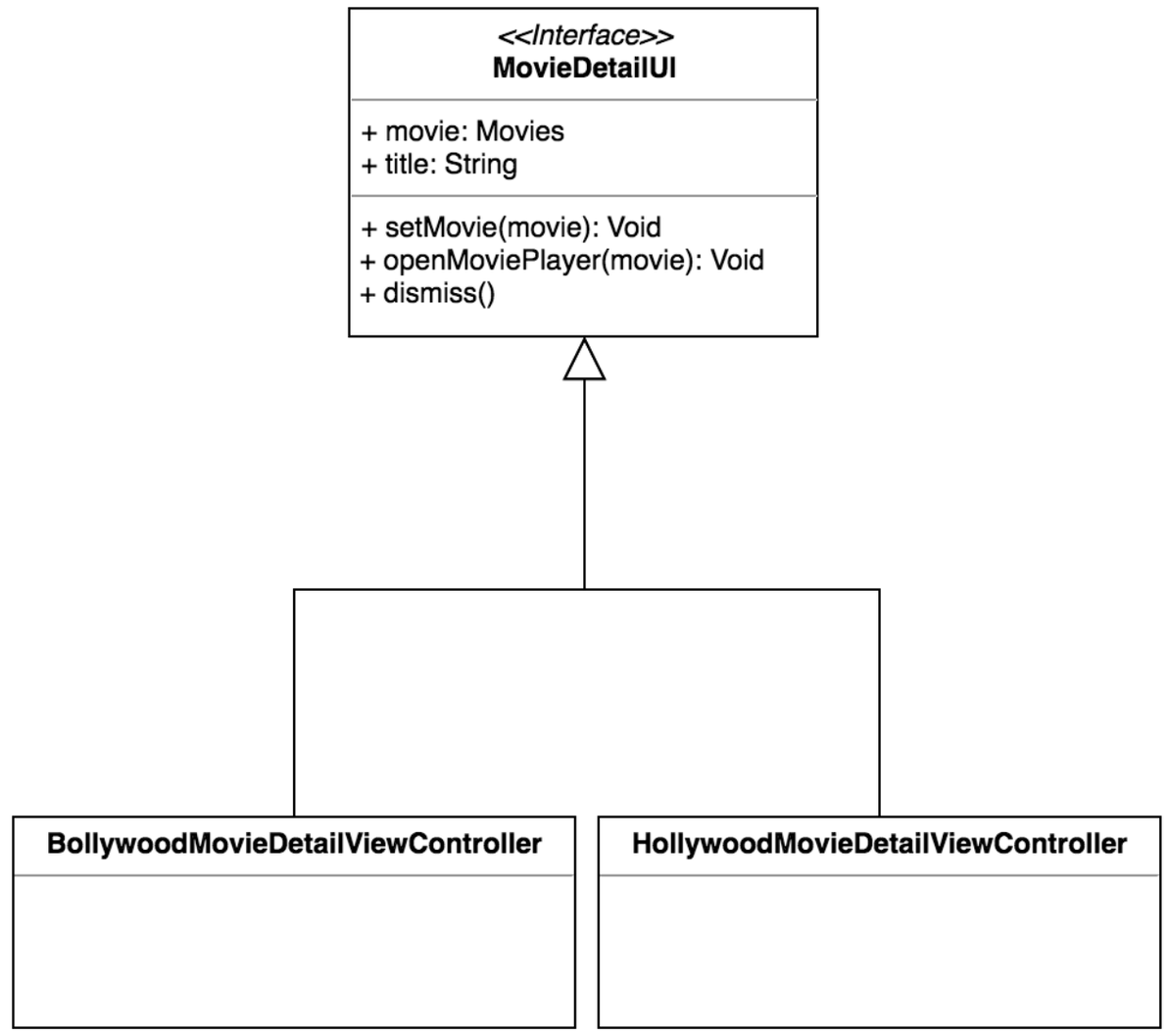
Using Swift Extensions to provide Dependency Injection

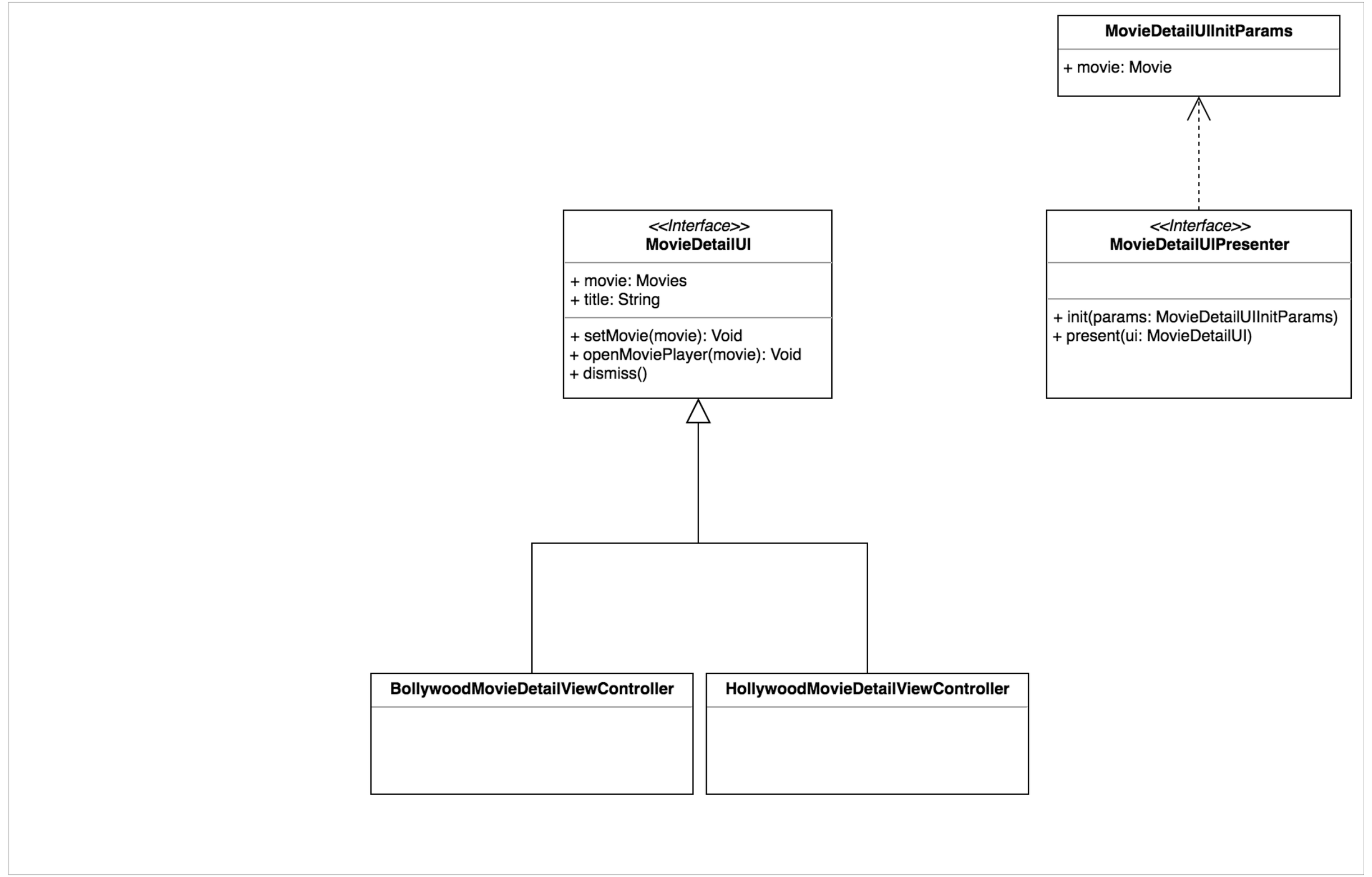
- In Swift, extensions allow us to extend any Type including Protocols.
- When a Protocol is extended using extensions, the implementing class of the Protocol gets all the extended functionalities too.

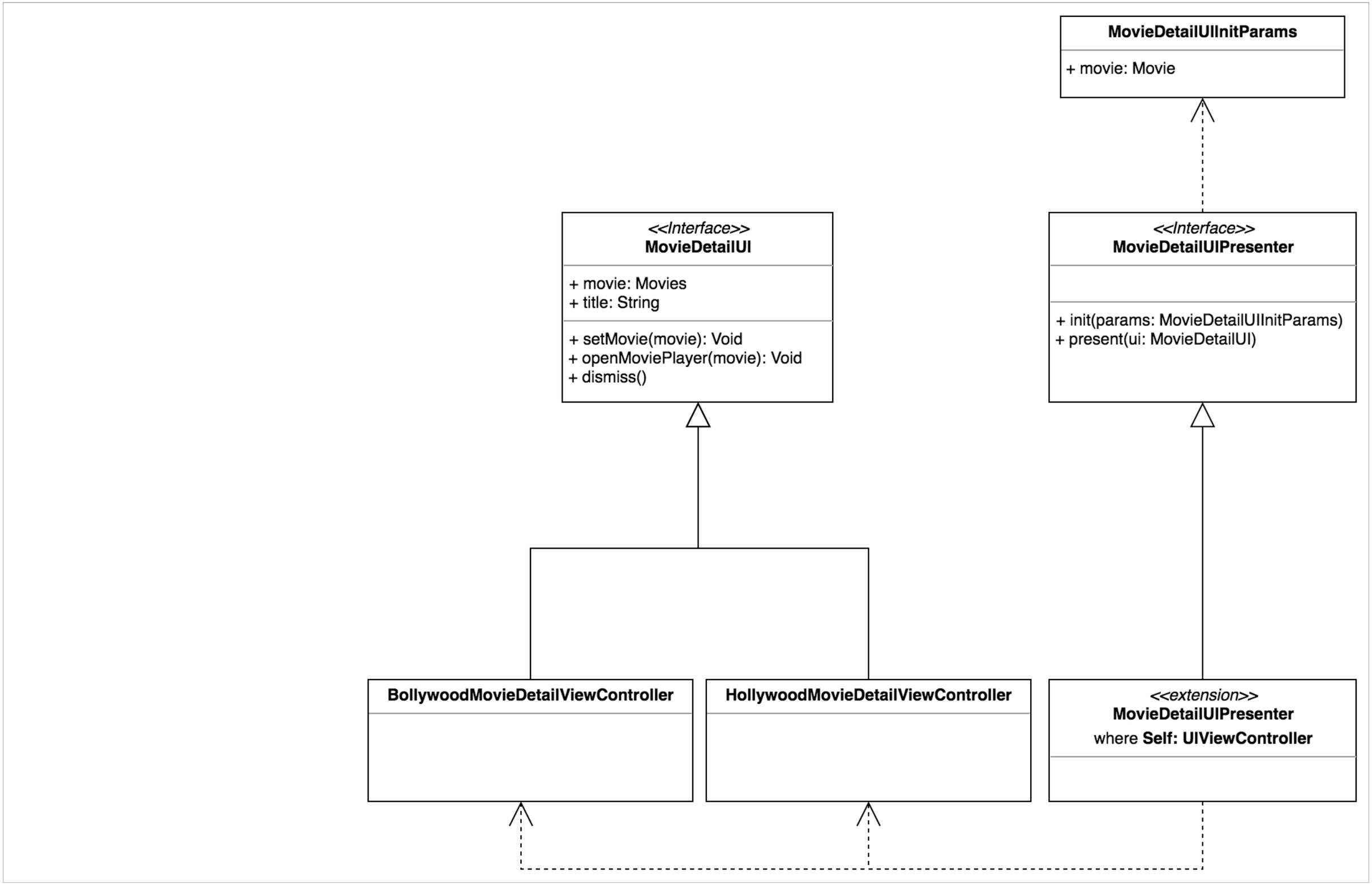
```
protocol Adder {
    func add(numbers: [Int]) -> Int
}

extension Adder {
    func add(numbers: [Int]) -> Int {
        var sum = 0
        for number in numbers {
            sum += number
        }

        return sum
    }
}

class Calculator: Adder {

}

let calculator = Calculator()
let sum = calculator.add(numbers: [3,5,1])
```

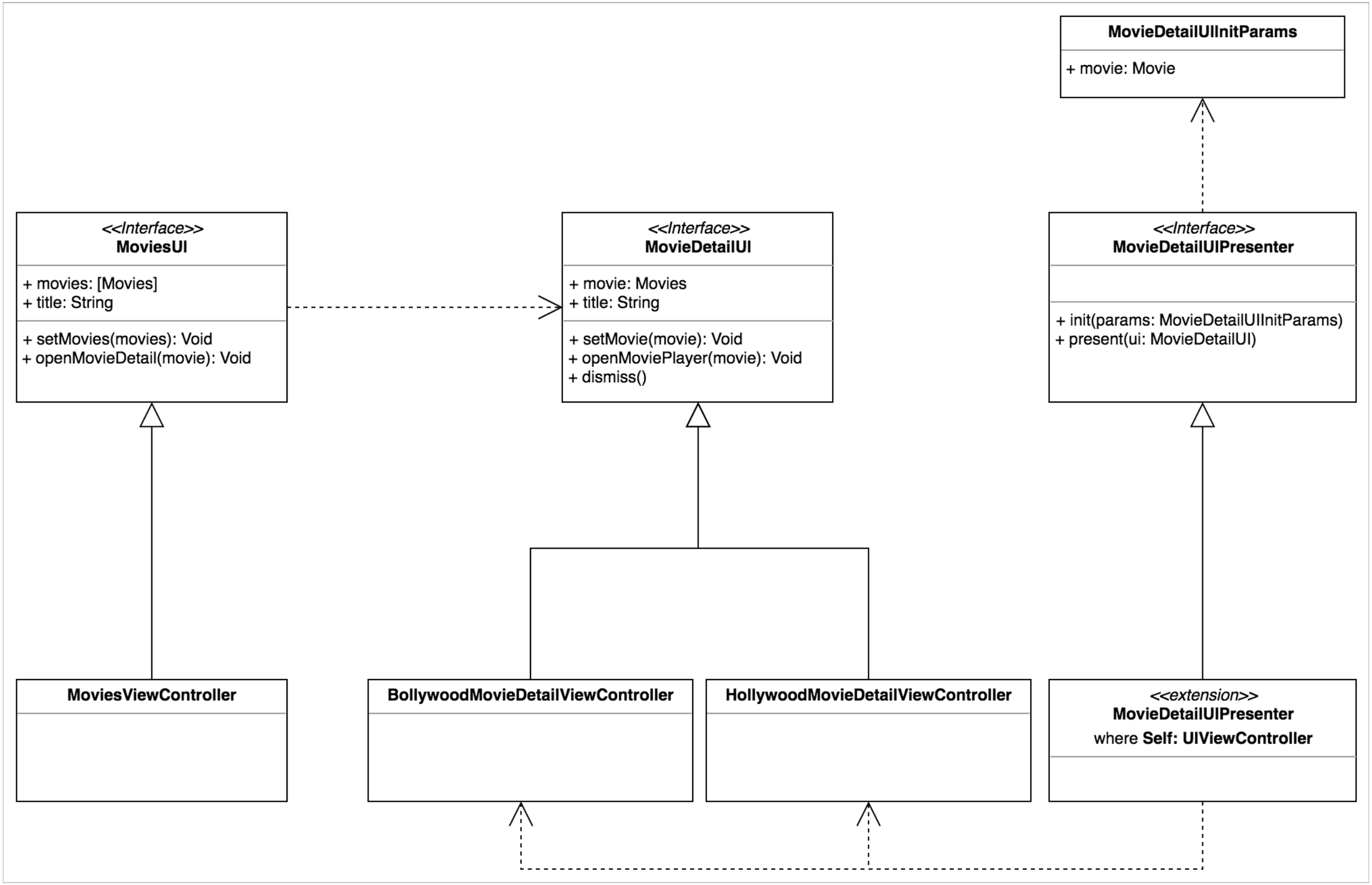
Using Swift Extensions to provide Dependency Injection

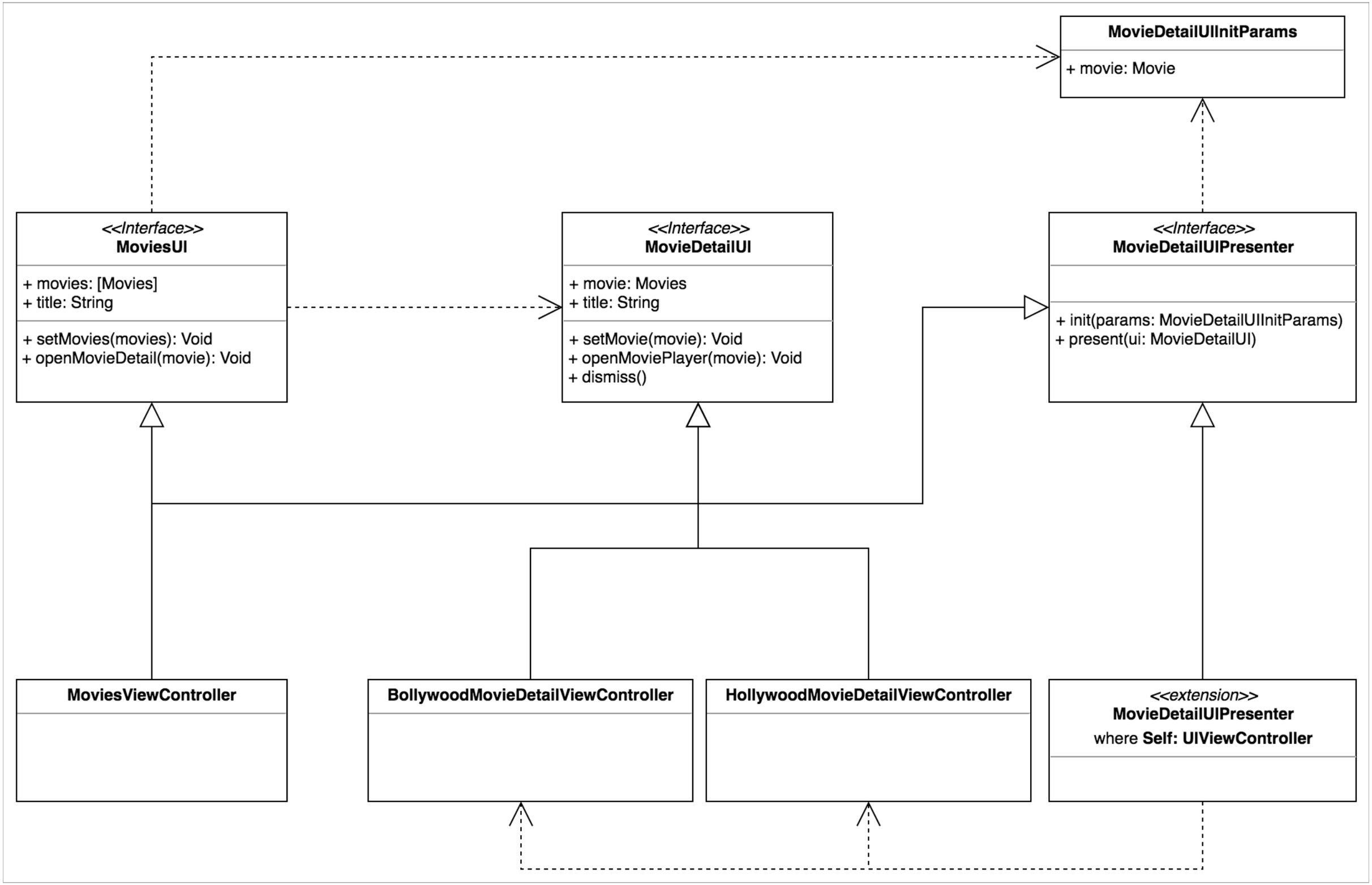

<<Interface>>
MovieDetailUI
+ movie: Movies
+ title: String
+ setMovie(movie): Void
+ openMoviePlayer(movie): Void
+ dismiss()

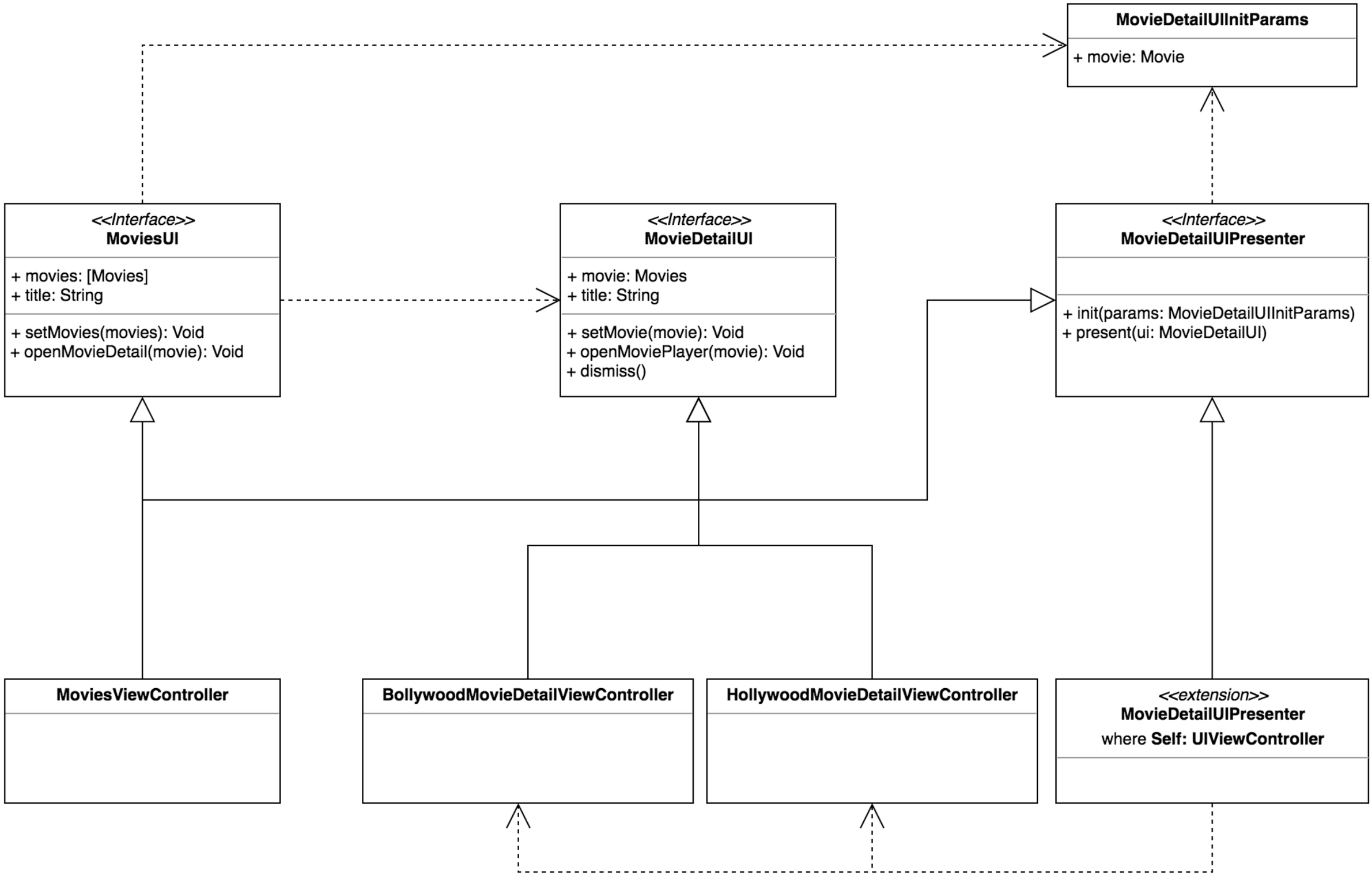

<<Interface>>
MovieDetailUI
+ movie: Movies
+ title: String
+ setMovie(movie): Void
+ openMoviePlayer(movie): Void
+ dismiss()
BollywoodMovieDetailViewController
HollywoodMovieDetailViewController


MovieDetailUIInitParams
+ movie: Movie
<<Interface>>
MovieDetailUI
+ movie: Movies
+ title: String
+ setMovie(movie): Void
+ openMoviePlayer(movie): Void
+ dismiss()
<<Interface>>
MovieDetailUIPresenter
+ init(params: MovieDetailUIInitParams)
+ present(ui: MovieDetailUI)
BollywoodMovieDetailViewController
HollywoodMovieDetailViewController


MovieDetailUIInitParams
+ movie: Movie
<<Interface>>
MovieDetailUI
+ movie: Movies
+ title: String
+ setMovie(movie): Void
+ openMoviePlayer(movie): Void
+ dismiss()
<<Interface>>
MovieDetailUIPresenter
+ init(params: MovieDetailUIInitParams)
+ present(ui: MovieDetailUI)
BollywoodMovieDetailViewController
HollywoodMovieDetailViewController
<<extension>>
MovieDetailUIPresenter
where Self: UIViewController


MovieDetailUIInitParams
+ movie: Movie
<<Interface>>
MoviesUI
+ movies: [Movies]
+ title: String
+ setMovies(movies): Void
+ openMovieDetail(movie): Void
<<Interface>>
MovieDetailUI
+ movie: Movies
+ title: String
+ setMovie(movie): Void
+ openMoviePlayer(movie): Void
+ dismiss()
<<Interface>>
MovieDetailUIPresenter
+ init(params: MovieDetailUIInitParams)
+ present(ui: MovieDetailUI)
MoviesViewController
BollywoodMovieDetailViewController
HollywoodMovieDetailViewController
<<extension>>
MovieDetailUIPresenter
where Self: UIViewController


MovieDetailUIInitParams
+ movie: Movie
<<Interface>>
MoviesUI
+ movies: [Movies]
+ title: String
+ setMovies(movies): Void
+ openMovieDetail(movie): Void
<<Interface>>
MovieDetailUI
+ movie: Movies
+ title: String
+ setMovie(movie): Void
+ openMoviePlayer(movie): Void
+ dismiss()
<<Interface>>
MovieDetailUIPresenter
+ init(params: MovieDetailUIInitParams)
+ present(ui: MovieDetailUI)
MoviesViewController
BollywoodMovieDetailViewController
HollywoodMovieDetailViewController
<<extension>>
MovieDetailUIPresenter
where Self: UIViewController


MovieDetailUIInitParams
+ movie: Movie
<<Interface>>
MoviesUI
+ movies: [Movies]
+ title: String
+ setMovies(movies): Void
+ openMovieDetail(movie): Void
<<Interface>>
MovieDetailUI
+ movie: Movies
+ title: String
+ setMovie(movie): Void
+ openMoviePlayer(movie): Void
+ dismiss()
<<Interface>>
MovieDetailUIPresenter
+ init(params: MovieDetailUIInitParams)
+ present(ui: MovieDetailUI)
MoviesViewController
BollywoodMovieDetailViewController
HollywoodMovieDetailViewController
<<extension>>
MovieDetailUIPresenter
where Self: UIViewController

MoviesUI

MovieDetailUI

MovieDetailUIInitParams

+ movie: Movie

<<Interface>>
MoviesUI

+ movies: [Movies]
+ title: String

+ setMovies(movies): Void
+ openMovieDetail(movie): Void

<<Interface>>
MovieDetailUI

+ movie: Movies
+ title: String

+ setMovie(movie): Void
+ openMoviePlayer(movie): Void
+ dismiss()

<<Interface>>
MovieDetailUIPresenter

+ init(params: MovieDetailUIInitParams)
+ present(ui: MovieDetailUI)

MoviesViewController

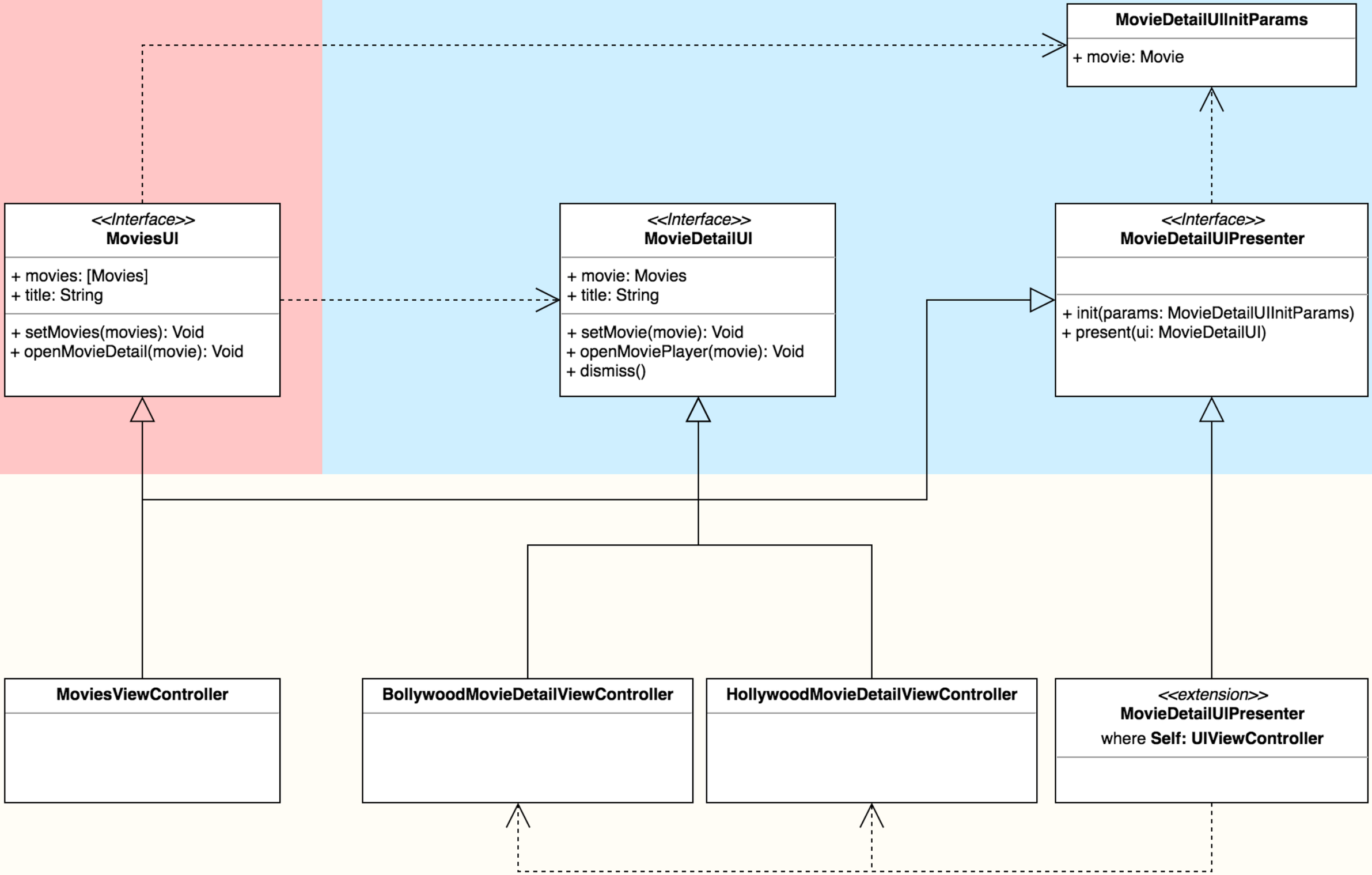
BollywoodMovieDetailViewController

HollywoodMovieDetailViewController

<<extension>>
MovieDetailUIPresenter
where **Self: UIViewController**

MoviesUI

MovieDetailUI



Demo

Questions?