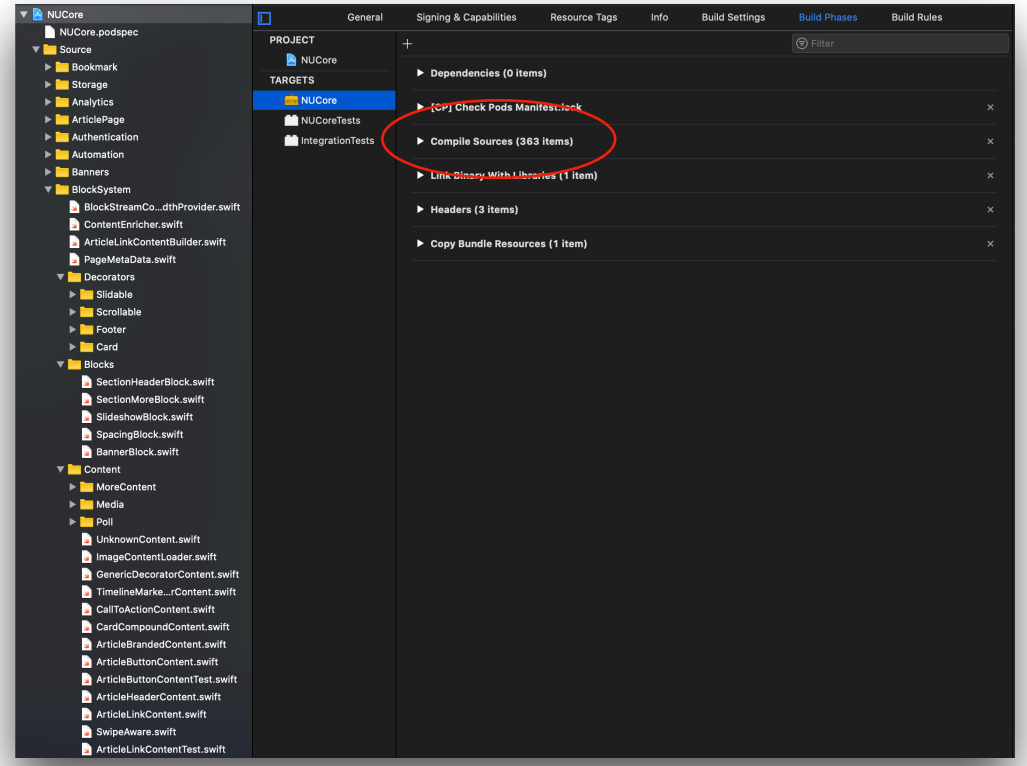
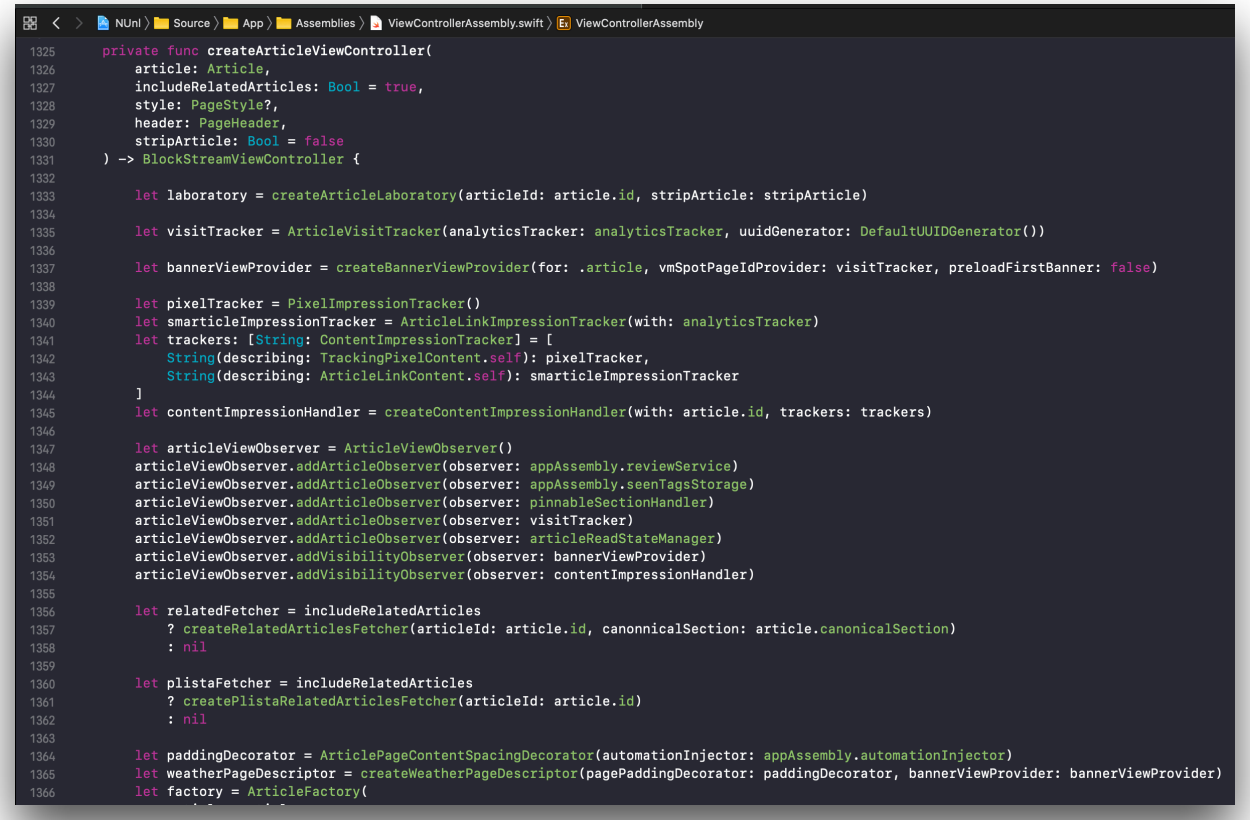
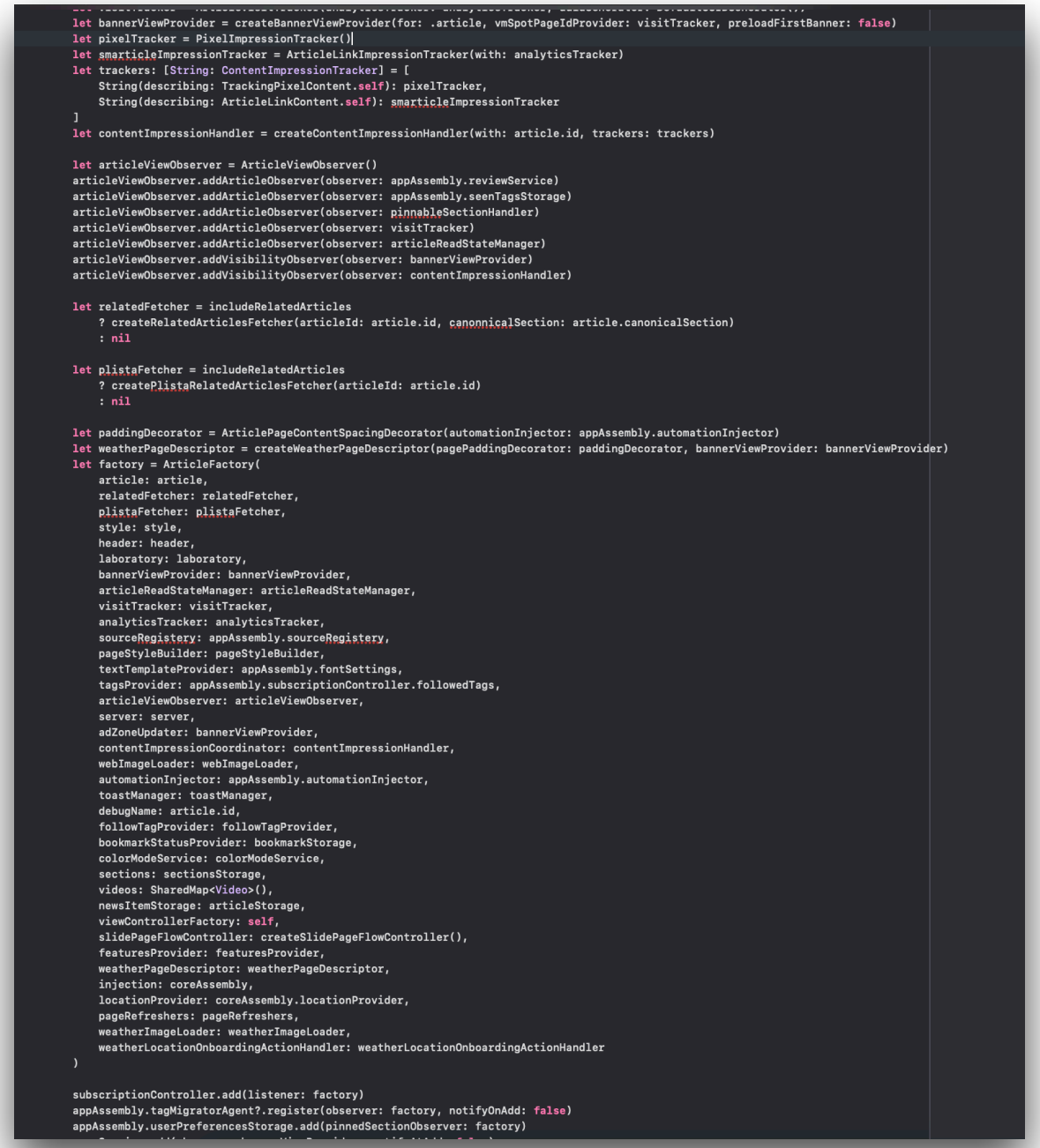
1. Modularisation

All architecture logic is coupled inside NuCore module(363 files) which is quite huge functionality in the app. All app dependencies are assembled in Application which is now 1750 lines of code. So we are quite tight to dependency that we have right now.



All dependencies for **ViewController** is assembled in **ViewControllerAssembly** which is now *1750 lines of code*. We are quite tight to dependency that we have right now.



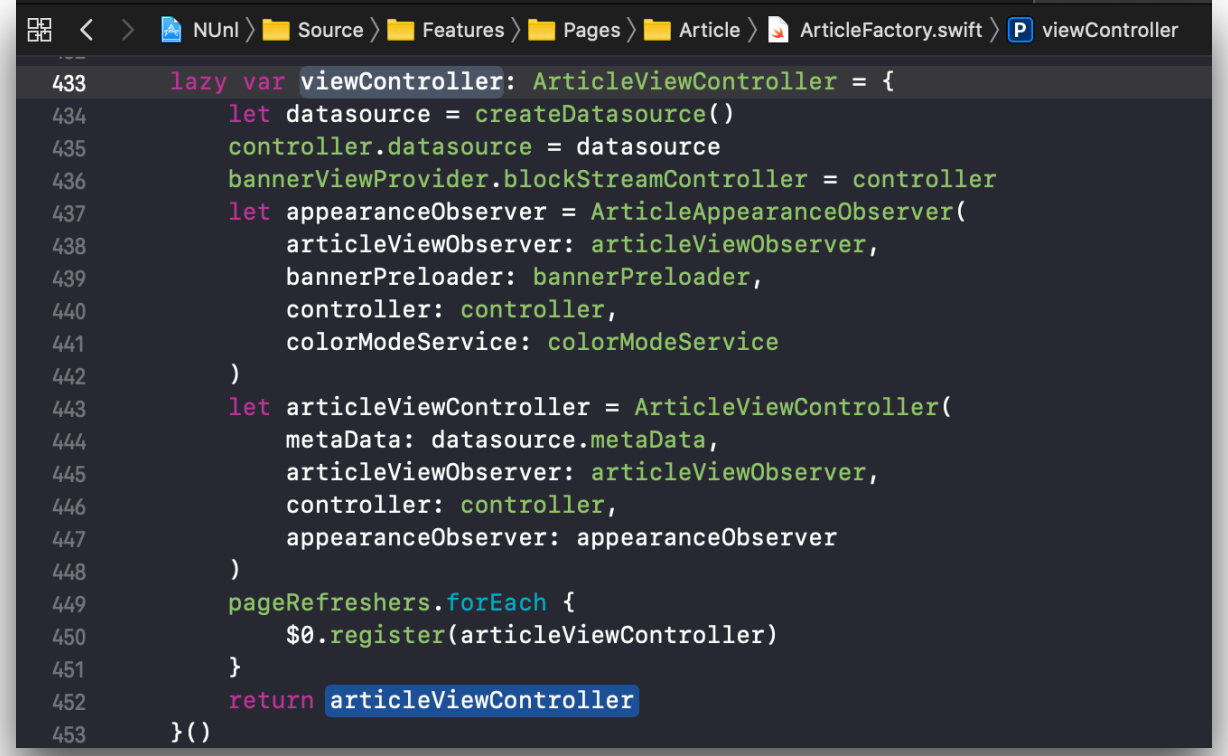


```
let bannerViewProvider = createBannerViewProvider(for: .article, vmSpotPageIdProvider: visitTracker, preloadFirstBanner: false)
let pixelTracker = PixelImpressionTracker()
let smarticleImpressionTracker = ArticleLinkImpressionTracker(with: analyticsTracker)
let trackers: [String: ContentImpressionTracker] = [
    String(describing: TrackingPixelContent.self): pixelTracker,
    String(describing: ArticleLinkContent.self): smarticleImpressionTracker
]
let contentImpressionHandler = createContentImpressionHandler(with: article.id, trackers: trackers)

let articleViewObserver = ArticleViewObserver()
articleViewObserver.addArticleObserver(observer: appAssembly.reviewService)
articleViewObserver.addArticleObserver(observer: appAssembly.seenTagsStorage)
articleViewObserver.addArticleObserver(observer: pinnableSectionHandler)
articleViewObserver.addArticleObserver(observer: visitTracker)
articleViewObserver.addArticleObserver(observer: articleReadStateManager)
articleViewObserver.addVisibilityObserver(observer: bannerViewProvider)
articleViewObserver.addVisibilityObserver(observer: contentImpressionHandler)

let relatedFetcher = includeRelatedArticles
    ? createRelatedArticlesFetcher(articleId: article.id, canonnicalSection: article.canonicalSection)
    : nil

let plistaFetcher = includeRelatedArticles
    ? createPlistaRelatedArticlesFetcher(articleId: article.id)
    : nil

let paddingDecorator = ArticlePageContentSpacingDecorator(automationInjector: appAssembly.automationInjector)
let weatherPageDescriptor = createWeatherPageDescriptor(pagePaddingDecorator: paddingDecorator, bannerViewProvider: bannerViewProvider)
let factory = ArticleFactory(
    article: article,
    relatedFetcher: relatedFetcher,
    plistaFetcher: plistaFetcher,
    style: style,
    header: header,
    laboratory: laboratory,
    bannerViewProvider: bannerViewProvider,
    articleReadStateManager: articleReadStateManager,
    visitTracker: visitTracker,
    analyticsTracker: analyticsTracker,
    sourceRegistery: appAssembly.sourceRegistery,
    pageStyleBuilder: pageStyleBuilder,
    textTemplateProvider: appAssembly.fontSettings,
    tagsProvider: appAssembly.subscriptionController.followedTags,
    articleViewObserver: articleViewObserver,
    server: server,
    adZoneUpdater: bannerViewProvider,
    contentImpressionCoordinator: contentImpressionHandler,
    webImageLoader: webImageLoader,
    automationInjector: appAssembly.automationInjector,
    toastManager: toastManager,
    debugName: article.id,
    followTagProvider: followTagProvider,
    bookmarkStatusProvider: bookmarkStorage,
    colorModeService: colorModeService,
    sections: sectionsStorage,
    videos: SharedMap<Video>(),
    newsItemStorage: articleStorage,
    viewControllerFactory: self,
    slidePageFlowController: createSlidePageFlowController(),
    featuresProvider: featuresProvider,
    weatherPageDescriptor: weatherPageDescriptor,
    injection: coreAssembly,
    locationProvider: coreAssembly.locationProvider,
    pageRefreshers: pageRefreshers,
    weatherImageLoader: weatherImageLoader,
    weatherLocationOnboardingActionHandler: weatherLocationOnboardingActionHandler
)

subscriptionController.add(listener: factory)
appAssembly.tagMigratorAgent?.register(observer: factory, notifyOnAdd: false)
appAssembly.userPreferencesStorage.add(pinnedSectionObserver: factory)
```

The above example of the creation of **ArticleViewController** function inside **ViewControllerAssembly**, shows how many dependency are injected just to create an article view controller, which doesn't even have all types of blocks that we want to show.

For example, given a **GenericViewController (GVC)** we will have to inject all classes which are responsible for presenting, handling user logic and routing for each particular block(element).

Therefore, we will end up with a huge constructor with all the dependency injected. I believe, that the current way of solving it on the app is through this **ArticleFactory**, which has a constructor with all those dependencies that ArticleViewController will need, then multiple setups are made, then it returns the ArticleViewController, which ***inherits*** from **BlockStreamController**.



It is hard to test or mock it, debug and maintain, due to that when we are going to be willing to create this controller or factory, it will require all those dependencies.

When we have a BFF code in place we will need to add even more logic to our app.

2. CellModelFactory

Currently, we have an “array" of all possible elements which can be shown(***Contents***). The image below shows **only** for ArticleViewController, but we have the same for **SectionController**, **WeatherController**, **SettingsController** and so on. Hence, this will grow up even more with every new Block.

```
@CellModelFactoryBuilder
private func createCellModelFactory() -> CellModelFactory {
    (Prototype(ArticleHeaderContent.self), articleHeaderBuilder())
    (Prototype(ArticlePinContent.self), articlePinBuilder())
    (Prototype(BannerContent.self), bannerBuilder())
    (Prototype(BodyTextContent.self), textContentBuiler())
    (Prototype(SummaryContent.self), summaryBuilder())
    (Prototype(ArticleBrandedContent.self), brandedContentbuilder())
    (Prototype(InvisibleScrollDepthContent.self), invisibleScrollDepthContentBuilder())
    (Prototype(CopyrightContent.self), copyrightBuilder())
    (Prototype(DividerContent.self), dividerBuilder())
    (Prototype(ExternalContent.self), htmlBuilder())
    (Prototype(FollowTagContent.self), followTagBuilder())
    (Prototype(ImageContent.self), imageBuilder())
    (Prototype(LinkContent.self), linkContentBuilder())
    (Prototype(MoreContent.self), moreContentBuilder())
    (Prototype(NUJijContent.self), nuJijContentBuilder())
    (Prototype(PublicationTimestampContent.self), timestampBuilder())
    (Prototype(QuoteContent.self), quoteBuilder())
    (Prototype(SectionHeaderContent.self, "tinted"), tintedSectionHeaderBuilder())
    (Prototype(SectionHeaderContent.self, "plain"), plainSectionHeaderBuilder())
    (Prototype(SectionHeaderContent.self, "normal"), sectionHeaderBuilder())
    (Prototype(ArticleLinkContent.self, "normal"), articleLinkBuilder())
    (Prototype(ArticleLinkContent.self, "related"), relatedArticleLinkBuilder(isLast: false))
    (Prototype(ArticleLinkContent.self, "related-last"), relatedArticleLinkBuilder(isLast: true))
    (Prototype(ArticleButtonContent.self), articleButtonsBuilder())
    (Prototype(SlideShowContent.self), slideShowContentBuilder())
    (Prototype(SpacingContent.self), spacingBuilder())
    (Prototype(TagContent.self), tagsContentBuilder())
    (Prototype(TextContent.self), textContentBuiler())
    (Prototype(TrackingPixelContent.self), trackingPixelBuilder())
    (Prototype(PostContent.self), postContentBuiler())
    (Prototype(VideoContent.self), videoContentBuilder())
    (Prototype(VideoPlaceholderContent.self), videoPlaceholderContentBuilder())
    (Prototype(CommentContent.self), commentContentBuilder())
    // should be
    (Prototype(WeatherForecastContent.self, "card"), weatherForecastContentBuilderWithMoreFooter(withCard: true))
    (Prototype(WeatherForecastContent.self), weatherForecastContentBuilderVanilla())
    (Prototype(WeatherForecastContent.self, "more"), weatherForecastContentBuilderWithMoreFooter(withCard: true))
    (Prototype(WeatherCurrentContent.self), weatherCurrentContentBuilder())
    (Prototype(WeatherCurrentContent.self, "card"), weatherCurrentContentBuilder(withCard: true))
    (Prototype(WeatherDayPartsForecastContent.self, "card"), weatherDayPartsContentBuilder())
    (Prototype(WeatherDayPartsForecastContent.self), weatherDayPartsContentBuilder(withCard: false))
    (Prototype(WeatherRainWindCompoundContent.self, "card"), weatherRainWindCompoundContentBuilder(withCard: true))
    (Prototype(WeatherRainWindCompoundContent.self), weatherRainWindCompoundContentBuilder(withCard: true))
    (Prototype(WeatherRainWindCompoundContent.self, "vanilla"), weatherRainWindCompoundContentBuilderVanilla())
    (Prototype(WeatherTodayContent.self, "card"), weatherTodayContentBuilder(withCard: true))
    (Prototype(WeatherTodayContent.self), weatherTodayContentBuilder())
    (Prototype(WeatherActivitiesContent.self, "card"), weatherActivitiesContentBuilder(withCard: true))
    (Prototype(WeatherActivitiesContent.self), weatherActivitiesContentBuilder())
    (Prototype(WeatherRainMapContent.self, "card"), weatherRainMapContentBuilder(withCard: true))
    (Prototype(WeatherRainMapContent.self), weatherRainMapContentBuilder())
    (Prototype(WeatherForecastMapContent.self, "card"), weatherForecastMapContentBuilder(withCard: true))
    (Prototype(WeatherForecastMapContent.self), weatherForecastMapContentBuilder())
```

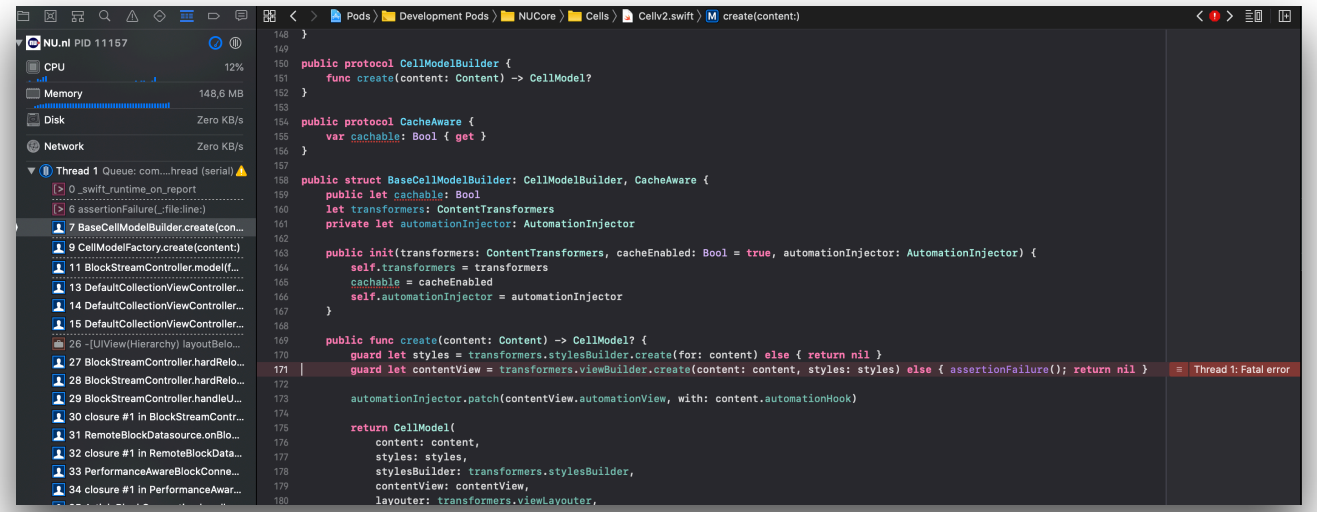
We don’t have a **strong types** defined. For example with **articleHeaderBuilder** *(see image below)* function we are trying to create styler, builder and layouter.

```
private func articleHeaderBuilder() -> CellModelBuilder {
    cellModelBuilder(
        stylesBuilder: ArticleHeaderContentStylesBuilder(),
        viewBuilder: ArticleHeaderContentViewBuilder(),
        viewLayouter: ArticleHeaderContentViewLayouter()
    )
}
```

Then, for **ArticleHeaderContentViewBuilder** *(see image below),* we have a type cast to some particular style:

```
public class ArticleHeaderContentViewBuilder: ContentViewBuilder {

    public init() {}

    public func create(content: Content, styles: ContentStyles) -> ContentView? {
        guard let styles = styles as? ArticleHeaderContentStyles, let content = content as? ArticleHeaderContent else { return nil }
        let contentView = ArticleHeaderContentView()

        contentView.label.attributedText = content.title.text(attributedText(styles.textStyles)
        contentView.label.isAccessibilityElement = true
        contentView.label.accessibilityIdentifier = "title_label"

        loadImage(view: contentView, content: content, styles: styles, aspectType: styles.imageAspectType)

        if let shadowStyles = styles.textStyles.shadowStyles {
            apply(textShadow: shadowStyles, to: contentView.label)
        }

        contentView.enableTap(with: HeaderTapAction(content: content))

        return contentView
    }
```

In case we have wrong class for style, we will just **return nil**, and after, have an **assertionFailure** in some other place. *Stack trace for assertion is on the left (See image below).*



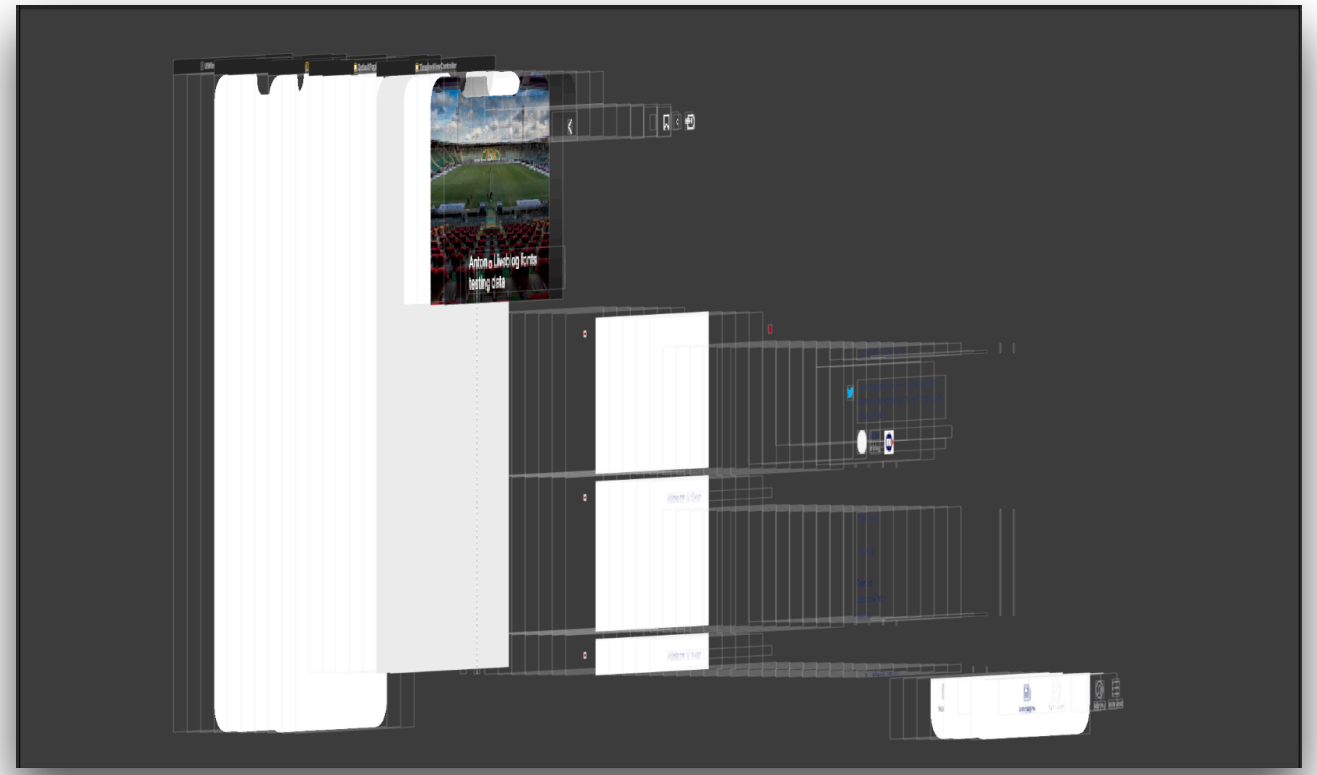
Therefore, we need to know here exactly the class of styler and a class of a content. For example for **50 blocks** I will need to know about **50 names of the Stylers**, **50 names of Contents** and **50 names of Layouters.**

This is functionality needed just to render a cell in view controller. As we can see, it's really hard to debug where I made a mistake if this type casting is failed.

We don't have a functionality which compiler can give us if we have for example array of enum cases. Then you will know it on compilation time, that you don't have a proper element layouter, builder or style.

3. Huge inheritance

Every subview is inherited from some class and changing in one class can lead to unpredictable behaviour in some other place. For example.



We have inheritance from **ContentView** all over the place and every new class inherited from. We can **override the functionality** and maybe some other view on the hierarchy might end up having unexpected behaviours.

```
public class PostContentView: ContentView, UITextNodeDelegate {
    public let tweetContainerView = UIView()
    public let textNode = UITextNode()
    public let postView = PostView()
```

```
open class ContentView: UIView, ContentViewType {
    public var onUserAction: ((UserAction) -> Void)?
    public var onVisibilityChanged: ((ContentVisibleEvent, Visibility, Visibility) -> Void)?

    public var tapAction: UserAction?
    public var contentVisibleEvent: ContentVisibleEvent?

    public private(set) var tapHandler: ((_ sender: UIControl) -> Void)?

    public private(set) var tapArea = ContentTapArea()

    private var clickableItems: [UIControl: UserAction]? = [:]

    // isAccessibilityElement on super view makes all its subviews inaccessible
    // which makes all system elements not accessible for Voice Over.
    // To solve this issue we create a small transparent view which holds block
    // specific automation accessibility identifier
    public let automationView = UIView()

    public func visibilityChanged(from: Visibility, to: Visibility) {
        if let contentVisibleEvent = contentVisibleEvent {
            onVisibilityChanged?(contentVisibleEvent, from, to)
        }
    }

    public override init(frame: CGRect) {
        super.init(frame: frame)

        addSubview(automationView)
        automationView.backgroundColor = .clear

        addSubview(tapArea)

        isUserInteractionEnabled = false
    }

    public required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    public func enableTap(with tapAction: UserAction) {
        self.tapAction = tapAction
        enableTap()
    }
```

4. Composability

We are quite tight to current implementation and it’s almost impossible for example to create a separate page not in our code without knowing about NuCore, ContentView or CellModelFactory, and so on.

5) Logic handling

Currently, logic inside **ArticleViewController** is handled by **ArticleFlowController** which is now more than *670 lines of code* with a lot of dependency injected which also can handle some functionality, or some actions. It's not even full Logic in app, we even have also logic for **IntermediateFlowController**, **SlidePageFlowController**, **TimelineFlowController** and for example onOpenComments*(see below).*

```
public lazy var articleFlowController: ArticleFlowController = {
    let presenter = self.tabBarController
    let tagsActionHandler = ArticleTagsActionHandler(
        with: presenter,
        subscriptionController: self.subscriptionController,
        toastManager: self.toastManager,
        colorModeService: colorModeService
    )

    let flowController = ArticleFlowController(
        injection: self,
        presenter: presenter,
        viewControllerFactory: self,
        storage: articleStorage,
        articleOptionsManager: articleOptionsManager,
        nuJijManager: nuJijManager,
        pinService: pinService,
        linkActionHandler: createLinkActionHandler(),
        tagsActionHandler: tagsActionHandler,
        shareController: shareController,
        bookmarkStorage: bookmarkStorage,
        slideShowFactory: createSlideShowFactory(),
        slideShowTracker: DefaultSlideShowTracker(analyticsTracker: analyticsTracker),
        weatherLocationOnboardingActionHandler: weatherLocationOnboardingActionHandler,
        trackerHandler: trackerHandler
    )

    flowController.onOpenSection = { [weak self] (identifier: PageIdentifier) in
        self?.sectionFlowController.openSection(identifier)
    }
    flowController.onOpenTag = { [weak self] (tag: Tag) in
        self?.sectionFlowController.openTag(tag)
    }
    flowController.onOpenComments = self.onOpenComments

    return flowController
}()
```

Logic about how comments are handled is not done by flow controller instead it's done by ***self*** which is **ViewControllerAssembly** *(1752 lines of code).*

```
private lazy var onOpenComments: (_ url: URL, _ title: String, _ source: String) -> Void = { [weak self] url, title, source in
    let analyticsEvent = GenericEvent(context: "article", action: "nujij", label: source)
    self?.analyticsTracker.track(event: analyticsEvent)

    let event = createOpenLinkEvent(url: url, external: false, title: title, options: [.gigyaBridge])
    self?.coreAssembly.eventRouter.route(event)
}
```

Missing argument for pa

Then, the logic of open comments *(see image above)* leads to **coreAssemble.eventRouter** who handles this event. **EventRouter** has an array of **5 different link handlers**, which can also handle this event.
There is no clear responsibility of who is actually responsible for handling one particular event.
If we are going to have hundreds different elements from BFF without clear responsibility of who is handling any type of logic we will end up in a huge messy code.

For example for Article page:

Tapping on link in text is handled by:

ArticleFlowController
-> EventRouter
-> OpenLinkAction
-> MainDeepleenkHandler
-> **Show next controller**

But, **Tap on Reactions** is handled by:

ArticleBarButtonsController *(instead of ArticleFlowController)*
-> ViewControllerAssembley
-> EventRouter
-> OpenLinkAction
-> mainDeepleenkHandler
-> **show next controller**

Tap on Image is handled by:
ArticleFlowController
-> **shows next controller**

Overall, we will show next controller but there are two different places who handle this and two different classes responsible for showing next controller.

Tracking for those events is handled some times on **ArticleFlowController** and others on **ViewControllerAssembly.**

6) Routing

The current status is that it's not clear where the routing take place on the app. We have MainViewController, TabBarController, DefaultAlertControllerPresenter who are responsible for routing in application. If I want to push or present something I need to inject presenter to be able to make navigation.

For example, **ArticleOptionsManager** this is injected in **ArticleFlowController** can also present something *(see image below)*:

```swift
final class ArticleOptionsManager: NSObject {
    typealias Injection = AnalyticsTrackerAware
    private let injection: Injection

    private let tooltipController: TooltipController
    private weak var alertController: UIAlertController?

    init(injection: Injection, tooltipController: TooltipController) {
        self.injection = injection
        self.tooltipController = tooltipController
        super.init()
    }

    // MARK: - Presenting menu
    func presentArticleOptionsMenu(_ presenter: Presenter, presentedFrom source: NUjijSource, article: ArticleMetaData, optionBuilders: [ArticleOptionBuilder]) {

        let batcher = ParallelClosureBatcher()
        var options: [ArticleOption] = []
        optionBuilders.forEach { builder in
            batcher.register { taskCompletion in
                builder.createOption(article) { option in
                    if let option = option {
                        options.append(option)
                    }
                    taskCompletion()
                }
            }
        }
        batcher.execute { [weak self] in
            self?.presentArticleOptionsMenu(presenter, presentedFrom: source, options: options)
        }
    }
```

It also can present something, and also knew about a presenter.

FlowController is also responsible not only for some business logic, like tracking, it's also responsible how to show ViewControllers. Breaking so the SOLID principles.

NUnl 〉 Source 〉 Features 〉 Pages 〉 Article 〉 ArticleFlowController.swift 〉 handle(articleLinkTapped:factory:)

Find articleLinkActionHandler

```swift
182     private func onAdClicked(clickURL: String, fallbackURL: String) {
183         guard let clickURL = URL(string: clickURL) else { return }
184         injection.eventRouter.route(createOpenLinkEvent(url: clickURL, external: false, fallBack: .external))
185     }
186
187     func containsFactory(with identifier: String) -> Bool { ••• }
193
194     func stopListening(factory: ArticleFactory) { ••• }
197
198     func handleClickTracking(action: UserAction) { ••• }
202
203     private func handleShowTracking(event: ContentEvent, from: Visibility, to: Visibility, factory: ArticleFac
216
217     func handleUserAction(action: UserAction, factory: ArticleFactory) { ••• }
264
265     // MARK: - Handlers
266
267     private func handle(action: LocationOboardingFlow, factory: ArticleFactory) { ••• }
272
273     private func handle(articleLinkTapped action: ArticleLinkTapped, factory: ArticleFactory) {
274         guard let segueAware = action.content as? LinkSegueAware else { return }
275         guard (segueAware.segue) != nil else { return }
276
277         // make sure read state of article are processed when returning to current page
278         factory.controller.resetData(ofType: ArticleLinkContent.self)
279
280         articleLinkActionHandler.handle(userAction: action, from: .article(factory))
281
```

7) Different architectural approaches

Overtime, different approaches mainly for **BlockStreamController** has been created, hence, you have different ways of interact with it, for example, with **RxSwift** *(See below on the right)* or with protocols/functions (See below on the left). We should have a team defined way of doing so, to be consistent over components.

```swift
// MARK: - AppearanceObserver

public protocol AppearanceObserver {
    func viewWillAppear(_ animated: Bool)
    func viewDidAppear(_ animated: Bool)
    func viewWillDisappear(_ animated: Bool)
    func viewDidDisappear(_ animated: Bool)
}

open class DefaultAppearanceObserver: AppearanceObserver {

    public let controller: BlockStreamController
    private let colorModeService: ColorModeService

    public init(controller: BlockStreamController, colorModeService: ColorModeService) {
        self.controller = controller
        self.colorModeService = colorModeService
    }

    open func viewWillAppear(_ animated: Bool) {
        colorModeService.applyActualColorModeBasedOnScheduledSettings()
        controller.isVisible = true
    }

    open func viewDidAppear(_ animated: Bool) {
        controller.handleViewBecome(visible: true)
    }

    open func viewWillDisappear(_ animated: Bool) {
        controller.isVisible = false
    }

    open func viewDidDisappear(_ animated: Bool) {
        controller.handleViewBecome(visible: false)
    }
}
```

```swift
open class BlockStreamController: NSObject {
    public let debugName: String
    public let config: BlockStreamControllerConfig
    public let collectionView: UICollectionView
    public let modelBuilder: CellModelBuilder?

    public var controller: CollectionViewController

    open private(set) var styles: BlockStreamControllerStyles
    open private(set) var cachedVisibleCells: [Cell] = []
    open var datasource: BlockDatasource?
    open var onDim: (((@escaping () -> Void)) -> Void)?
    open var onUndim: (() -> Void)?
    open var blocks: [Block] = []
    open var batchCount = 0

    var resumeType: ResumeType?

    // MARK: - Rx
    public var activityInProgress: Observable<Bool> { activityInProgressRelay.asObs
    private let activityInProgressRelay = BehaviorRelay<Bool>(value: false)

    public var isVisible: Bool {
        get { visibilityRelay.value }
        set { visibilityRelay.accept(newValue) }
    }
    private let visibilityRelay = BehaviorRelay<Bool>(value: false)

    public var onRefreshingDone: Observable<Void> {
        refreshingDoneRelay.asObservable()
    }
    private let refreshingDoneRelay = PublishSubject<Void>()

    public var onViewDidAppear: Observable<Void> {
        visibilityRelay
            .asObservable()
            .distinctUntilChanged()
            .filter { $0 }
            .map { _ in }
    }
```

8) Magic/Hardcoded UI (Strings, font size, font types, etc.)

Along the app we have hardcoded on multiple places different sizes for the font. Would be ideal to align with UX, and defined some names or categories for those font sizes. Same applies to font type, font weight, etc.

```
func cssSettings(_ size: FontSize) -> ParagraphExcerptHeadingSettings {
    _ = UIApplication.shared.preferredContentSizeCategory

    let fontWeight = fontContrast ? 400 : 300
    switch (deviceClass, size) {
    case (.iPhone, .normal):
        return ParagraphExcerptHeadingSettings(
            paragraph: FontInfo(fontSize: 15, lineHeight: 24, marginBottom: 14, fontWeight: fontWeight),
            excerpt: FontInfo(fontSize: 16, lineHeight: 25, marginBottom: 32, fontWeight: 500),
            heading: FontInfo(fontSize: 15, lineHeight: 25, marginBottom: 6, fontWeight: 600))
    case (.iPhone, .large):
        return ParagraphExcerptHeadingSettings(
            paragraph: FontInfo(fontSize: 16, lineHeight: 25, marginBottom: 15, fontWeight: fontWeight),
            excerpt: FontInfo(fontSize: 17, lineHeight: 25, marginBottom: 34, fontWeight: 500),
            heading: FontInfo(fontSize: 16, lineHeight: 25, marginBottom: 8, fontWeight: 600))
    case (.iPhone, .extraLarge):
        return ParagraphExcerptHeadingSettings(
            paragraph: FontInfo(fontSize: 17, lineHeight: 27, marginBottom: 16, fontWeight: fontWeight),
            excerpt: FontInfo(fontSize: 18, lineHeight: 27, marginBottom: 36, fontWeight: 500),
            heading: FontInfo(fontSize: 17, lineHeight: 27, marginBottom: 10, fontWeight: 600))
    case (.iPad, .normal):
        return ParagraphExcerptHeadingSettings(
            paragraph: FontInfo(fontSize: 17, lineHeight: 27, marginBottom: 16, fontWeight: fontWeight),
            excerpt: FontInfo(fontSize: 18, lineHeight: 28, marginBottom: 32, fontWeight: 500),
            heading: FontInfo(fontSize: 17, lineHeight: 28, marginBottom: 8, fontWeight: 600))
    case (.iPad, .large):
        return ParagraphExcerptHeadingSettings(
            paragraph: FontInfo(fontSize: 18, lineHeight: 28, marginBottom: 17, fontWeight: fontWeight),
            excerpt: FontInfo(fontSize: 19, lineHeight: 30, marginBottom: 34, fontWeight: 500),
            heading: FontInfo(fontSize: 18, lineHeight: 30, marginBottom: 10, fontWeight: 600))
    case (.iPad, .extraLarge):
        return ParagraphExcerptHeadingSettings(
            paragraph: FontInfo(fontSize: 19, lineHeight: 30, marginBottom: 18, fontWeight: fontWeight),
            excerpt: FontInfo(fontSize: 20, lineHeight: 31, marginBottom: 36, fontWeight: 500),
            heading: FontInfo(fontSize: 19, lineHeight: 31, marginBottom: 12, fontWeight: 600))

    }
```

```
//TODO: Adapt to the dynamic font change
func create(for content: Content) -> ContentStyles? {
    let styles = TextContentStyles()
    styles.textNodeStyles.textStyles.font = UIFont.lightFont(15) //fontProvider.getParagraphFont(with: fontSize())
    styles.textNodeStyles.textStyles.color = UIColor.primaryText
    styles.textNodeStyles.textStyles.lineSpacing = 4
    styles.textNodeStyles.textStyles.richFont = .default
    styles.textNodeStyles.textLinkColor = UIColor.mode.linkColor(with: tintColor)
    styles.textNodeStyles.contentPaddings = .zero
    styles.textNodeStyles.textStyles.topCorrectionStyle = .none
    return styles
}
```

No type safety instead of using enum where with statements are exhaustive which ensures that enumeration cases aren't accidentally omitted.

```
(Prototype(WeatherRainMapContent.self, "card"), weatherRainMapContentBuilder(withCard: true))
(Prototype(WeatherRainMapContent.self), weatherRainMapContentBuilder())
```

UI

UI elements can't be rendered without knowing about whole app architecture setup

Solved with a proposal

1) **Modularisation** - Every function part is a testable separate target which can be easily move to another workspace or project. (Mapper, BFF, Tracker)
2) **CellModelFactory** - Solved with type safe enum cases. (Try to add new bff element). Compiler will do everything for you. Easy onboarding process.
3) **Inheritance** - No inheritance. No UI inheritance. Ui built in a separate target. All Ui elements is dummy element with just Input (information what needed to render a view). Made with layers.
4) **Composability** - All modules can be easily tested. Or there is even a separate App where we can just inspect how UI element looks like (CardBoard app)
5) **Logic handling** - All logic is handled inside interactor. Even if we want to inject some dependecy we will know where to find a place where logic is handling.
6) **Routing** - Routing events received unidirectional from presenter. Or we can have some GlobalRouter who will handle not only viewcontroller event, like push notifications taps.
7) **Different architectural approaches** - One VIP architecture. Reactive code can be presented in interactor and not distributed over the place. For example receive event of viewControllerDidAppear combine it with BannerPool ready event. Update presenter with new add.
8) **Magic UI** + **UI** - all UI elements is in separate module. Can be easily shown in app. See CardBoard app
9) **Magic Strings** - everything is type safe. Try to uncomment BFFFetcher -> fetchAll(). Everything is coupled using enums which ensures that enumeration cases aren't accidentally omitted. You will receive just one assertion failure in Mapper, not even in an app. So our app will not crash if new element will be pushed from BFF.
10) **Clear way how to add new content**. Try step 9
11) Easy onboarding.