Feature Management Using YAML Markups

Introduction and Objectives

Any product evolves over time. **Grosser requirements become more and more fine grained**. In turn, these impact the coding, UI development, testing. We often don't even have visibility of how much cost it will be in terms of estimation of cost, development time and so on.

In contrast, non-functional requirements include performance, security, reliability, maintainability, safety and so on. These are identified and managed upfront, and don't really change much during the lifetime of a project.

Particularly for startups, initial iterations of a product are very crucial: You have limited budget, and yet, with every iteration, you want a functional product that can actually be used for real

work. The startups either use mockups, or user interfaces themselves as the feature documentation.

To help define a good feature development/tracking process, we identified these following advantages, if we were to adopt a special YAML markup to track features and related artifacts of a product:

Document features/sub-features as they are thought of, and later, detailed out. Features start off as gross. For e.g. “Implement search functionality”. And then, we detail it out. Do we want to search only titles for now? And later add content? Do we want auto-completion as we type? And so on. These form a feature hierarchy, which would have to be captured in our markup. In the absence of such a markup, one tends to simply use requirement specification documents, and we can’t really use structured information then.

Track associated artifacts. Particularly UI artifacts, such as mockups. For e.g. if there is a mockup of home screen, how will it look like after we add search bar? As UI developers develop these screens, our markup will have to track them, associated with relevant subfeatures. This will also apply to other product development processes: Code, Tests and so on. It should be possible for us to find out precisely that code which is used to implement a particular sub-feature.

Create a communication model to track TODO items. A lot of fine details often need to be created and communicate when a feature is being implemented either in mockup, or it is being coded, or tests are being developed. The experience of senior and experienced developers needs to be captured. For e.g. in our search example, the instruction for developer might be “Search autocomplete response should be within a sub-second latency.” Or, to UI developer, it might be, “Autocomplete suggestion box should only list 5 items”. This communication, which is intended to reflect in artifacts associated, is best captured and associated at sub-feature level, and our markup should facilitate that.

And finally, auto-generate the artifacts from features when frameworks support it. Let us say, our UI developers are directly using frameworks such as bootstrap (or even React native, Vue/Vuetify etc) to create mockup UI with example data. So, if I want to add a search bar in the main notification tab of the home screen for the search feature, it could be hand edited and seen in a browser. But it is also possible to do this in automated manner: Just make the main notification tab take a list of components from a configuration file. Then, all we can do is to develop a simple approach to add our search component definition to this configuration file automatically. So in theory, the feature definition can now drive modification of UI at configuration time. (Tools such as stencil are designed to help in this approach.) This approach can not only be applied to UI artifacts, but also to development artifacts. For e.g. we may have a pluggable architecture and a feature may define its own plugin. (Note that we are not talking of

code generation here. We are only saying, the code/configuration is written independently, and triggered to be included in the main codebase upon inclusion of this feature.

Tooling around Feature and Various Artifact YAMLs

Given the YAML Markup, we can then ease some of the product development tasks:

- **Generate Feature Documentation from specs - In form of Excel etc.** Or, we can directly generate Wiki pages. We can even generate requirements documents.
- **Tracking and linking artifacts.** Every feature and sub-feature would have to get its own ID. This helps us create appropriate linkages and traceability. More important, it will give us a very quick access to relevant contributed artifacts from various teams specific to a sub-feature.
- **Planning and Tracking progress.** While planning and tracking happens in different tools, the process gets very structured by having programmatic access to feature/subfeature and artifact YAMLs.
- **Autogenerate Artifacts.** When we use appropriate frameworks, we should even be able to autogenerate artifacts related to specific sub-features.

Feature Management YAML Vs. UML and other Visual Approaches

UML is popular means of doing various visual documents to convey the intent of the applications. It is best suited for interactions (Use cases, Interactions via sequence diagrams, code organizations via component diagrams) and so on.

However, UML is a specification, and carries with it a lot of semantics, which needs to be learnt to properly use it as a collaboration medium. Over the time, it is more used as a communication tool mainly for documentation purposes, rather than intended purpose of documenting object models, classes and their relations, which can then be used to generate code.

Recent hacker news thread has elaborate discussions on this: Why UML “Really” died? .

The intent of our YAML approach is **to create an incremental process** to capture the requirements (and in particular features), and then to connect them to changing UI artifacts, Development artifacts and Test artifacts. A change must connect to the change. For example, an UML sequence diagram may be modified due to feature requirement. So we want the change to be identified in feature, and the changed subfeature will now link to new UML diagram.

Let us say we are using a markup (such as PlantUML) to draw UML sequence diagram. In such a case, the change in markup itself is captured as part of UML change, and then used to create a diagram as a dynamically created, on-demand, picture.

Therefore, irrespective of tools and frameworks used for different aspects of product development, we can create a markup approach that documents a change.

How do teams currently manage features?

Feature-driven development was a development methodology from 1999, focused on features (and sub-features) to drive the development. In contrast, scrum and other agile methodologies focus on incremental development of (usually) fixed duration to advance the projects. As a result, they tend to use project collaboration tools to also document feature implementations:

- Maintaining a set of documents detailing feature requirements. And using issue tracking tools to manage them - i.e. communicate and track with developers.
- Using collaborative project management software as a proxy for feature management as well. For e.g. agile development tools have sprint management capabilities to track user stories to completion. For e.g. Trello is often used for idea management using its boards concept.

Dedicated feature management tools like zepel (https://zepel.io/) boldly point out that features shouldn't be managed as issue tracking.

Also, there is now well recognized design pattern of **Feature Flags** - Architect a product to enable/disable a feature at run time, depending on say, subscription option chosen. This also needs features to be recognized separately.

In general, it seems like feature management as a separate discipline within a product development hasn't taken roots. Our approach, if done well, could change that.

YAML Specifications Summary

We believe there is a need for following types of specifications. A project may of course only start with features YAML specs and use others as an option.

Type of YAML Markups	Main information	Associations
Features YAML Files	Capture feature info, assign an ID. And store the feature status - Whether it is just an idea, accepted for implementation, planned, implemented and so	Association of Ownership Association of dependency (On artifacts of UI / App / Test Modules)

	on.	
UI Artifacts YAML Files	Capture information about UI artifacts such as mockups, screens etc. It could also refer to specific, added/removed changes in mockups and screens.	
Code Modules YAML Files. YAML files may also be specifically identified portions within modules/files. See the details.	System, Components and Linkages between them, with IDs assigned. (C4 Model, see in references)	
Test Specification YAML Files	TODO	

Please note that the project planning, milestones and the like are NOT the focus of these YAML files; they would be managed in traditional tools.

YAML for feature/subfeature hierarchy

Feature may initially be just by itself, with only a loose description:

```
ID: SearchInGroupChat
Feature: Search in group chat
DependsOn:
   - GroupChat
Description: |
       Provide a menu item “Search” for group chat. Clicking it should
       result in a new screen with search results from chats of the
group.
SubFeatures:
   - ID: AutoCompleteSearchInGroupChat
     Status: Planned
   - ID: SearchOnlyTitles
     Status: Planned
```

And later, we make it refined, but adding sub-features, each of which may be an independent aspect, but depending on already having some existing features. So our markup should capture the definitions and dependencies. Examples:

- Search is a typical feature in any product.
    - Adding autocomplete is a subfeature.

    - Adding an option to search only in titles is a sub feature which is meaningful in context of main feature, and is orthogonal to the autocomplete subfeature.
- In whatsapp, “Mute notifications” is one of the subfeatures under group chat. Within that, “Mute notifications during night” may be sub feature of Mute Notifications.

So our markups will be as follows (for Autocomplete):

```
ID: SearchAutocompleteInGroupChat
Feature: Search Autocomplete in group chat
DependsOn:
   - SearchInGroupChat
Description: |
      As user types, provide a means to show list of chats starting
with
      that word.
```

We will later on see how to augment the related artifacts and modules with these definitions.

Please note that whenever we say “Feature”, it may mean “SubFeature” also, because every feature is sub feature of another feature.

How and when are other artifacts referenced in a feature

As seen above, we need to capture dependencies on the other artifacts.

Ownership of Artifact. We capture if a feature totally owns an artifact fully i.e. the artifact is otherwise not owned by any other feature. We can do this at any level of subfeature. Note that the artifact is provisioned (i.e. ID is created with some details), but the end artifact - such as mockup or screen, or a development module, is not yet created.

Dependency on Artifact. Dependency is when we require an artifact which may be owned by a parent feature, or even another feature. And this feature is only interested in changes to this artifact. These changes may be applied manually on a copy of that artifact, or may be applied automatically (within some sort of framework). If a new artifact is created manually, it must be managed within that artifact definition YAML, and reevaluated if the parent’s artifact changes.

Actions available from artifacts

Consider the following feature spec.

```
ID: SearchAutocompleteInGroupChat
Feature: Search Autocomplete in group chat
Description: |
     As user types, provide a means to show list of chats starting
with
      that word.
Owns:
   - UI_SearchAutocompletionDropdownBlock
   - Function_SearchAutoComplete
DependsOn:
   - UI_SearchInGroupChat
   - Code_SearchModule
Actions:
   - UIAction_ShowAutocompleteBlock
           Block: UI_SearchInGroupChat
           EnableCapability: AutoCompleteWidget
           AssociateWith: UI_SearchAutocompletionDropdownBlock
           # Enable an already existing capability to be used for our
           purpose
   - CodeAction_AddFunctionToModule:
           Module: Code_SearchModule
           Function: Function_SearchAutoComplete
           # Add this function to the said module.
```

Here, we can see that this subfeature owns two artifacts. One is for UI, another is for code related to search module. That means, these two artifacts are created solely for the purpose of this subfeature.

The feature also has dependency on existing artifacts, which are also listed here by the feature. First one is on UI. We want to modify the existing search UI to add a autocomplete feature. This is done by two means as follows:

Manual Action. If the UI involves mockups, then it is treated as just a documentation action to communicate to UI implementer: “Modify the existing search box to also show dropdown autocompletion box whenever something is typed.”

Automated UI framework actions. Let us say, our UI is being done using bootstrap UI framework, with some example data. This framework, as we know, provides readymade search widget. With right options in markup, it also implements autocompletion as well.
Then, all we want is that a script can automatically enable this widget to do autocompletion also. Of course, we should also supply an implementation of how the drop down block should look like (also in bootstrap). So our feature will simply trigger this action. Needless to say, such list of actions should already be available to us within the bootstrap framework. Which means, the framework will adopt and enhance with such actions, if everyone starts using this markup approach.

Code Module Actions. The code modules and individual functions, or configuration files can also allow automated modifications. For e.g. a command line parser module may allow a configuration file to be modified to add new command line options specific to features. These actions need to be established a-priori by the code framework team. The specs should specify how to specify such actions.

Please note that our primary intention is to establish what "work" needs to be done to implement the feature. In UI, In Code, and in the Tests. The intention can be expressed for documentation only i.e. a senior architect creating the set of actions for coders to implement (and not to miss anything!). Or they can be automated and used by framework to automatically add code or configurations.

We try to document the type of actions that one could perform, and therefore, associate with features to give a better idea.

UI Artifact allowed actions

Artifact Type	Artifact Sub Type	Artifact Allowed Action	Arguments	Meaning
UI	Screen Block	Insert a Block at specific location in this screen	Screen's ID, Position word	Insert this block at the top, or bottom etc. These positions are pre-identified within the Screen block.
	Menu Block	Insert a block at specific location in this menu	Menu's ID, Position word (top, bottom, number)	Block can be just a word, or a small widget such as on/off.
	Toolbar	Allows to insert a block in specific position in	Toolbar ID, Position	Can be, e.g., icon, or a search input block, etc.

		toolbar		
	Settings Menu	Insert a new Setting block	Setting ID, Setting Sub Block ID, Position,	Can be a mini screen.

Development Code Artifacts allowed actions

Artifact Type	Artifact Sub Type	Artifact Allowed Action	Arguments	Meaning
Configuration Module	Named Configuration	Add a new configuration element	Config block ID	Add feature specific configuration element. For e.g. command line options. Host/port options, and so on.
Modules Home Directory	Module directory	Create a module directory, populate with specific template if any.	Module name, Name of file Name of template	Creates a new module. For e.g. all authentication modules are undr Authentication directory. A particular auth mechanism (a feature) instructs particular module to be created under this directory.
	Plugin Module	Add a plugin module of some type	Type of Plugin	All plugins adhering to specific registration semantics, and provide uniform API.
A File in Module	N/A	Add a function to file	Name of function, path of codebase	It should not matter where the function is added. But annotate it so that we identify it belongs to this feature.
	An existing function	Add a single (or few lines) of code that is related to this feature,	At pre-identified location in this function.	An example is a function that calls a series of initializations, each for specific feature. Ordering can be managed by positions, if it is important. Ideally it is used when ordering is not important.
Data Modeling	DB	Create a Table or Add a column to table	Specifics of table/columns.	Provide information to add tables and/or columns to tables.

Note that a single feature may trigger multiple actions.

Some examples of actual use in applications:

- Insert a few lines of code (which are themselves not functions) within an existing function.
    - For e.g. Initialization calls managed by dynamically inserting functions in a startup/shutdown code.
    - Or, one may do a series of checks, each specific to feature within a function. For e.g. “add a user” feature may request code to be added to verify that userid meets the specifications provided. This code may be managed as a YAML artifact, and be inserted into a specific useradd function dynamically, at a

specified location within the function - where it allows various checks to be made before calling database add.
- Add configuration elements specific to feature:
    - For e.g. File paths to store the files
    - Command line options
- Manage URL Paths of application (in web framework): Insert the URL Handler function as well as path registration..
- Use a tool to manage database schema. Drive the schema addition from features i.e. let feature adds required columns / tables etc.
- Manage plugins which all share the same function signatures / APIs. Insert and manage them through features.

YAML for managing UI Artifacts

Features will be associated with UI artifacts as they are being developed in parallel by UI designers. We should associate these artifacts such as mockup screens (or fully developed UI) with features to ease tracking and access.

Various mechanisms are used to capture the UI: Hand drawings, Mockup screens created by tools, Use of bootstrap-like widget system, High resolution tools like Figma and so on. Our approach should be orthogonal to these - with the intention to mainly track feature specific artifacts, or sub-artifacts.

Terminology / Glossary

- **Block.** We use the generic term “Block” for an UI artifact, since it owns space on the screen real estate. It is always an image for our purpose, irrespective of tool.
- **E-Blocks.** A feature/sub-feature may totally own a mockup screen. For e.g. mute notifications may have its pop up screen totally developed and used for this feature. We call such artifacts as E-Blocks, E meaning “End artifact”. E-Blocks are either managed by end user (i.e. different versions are associated with features as they evolve)
- **F-Blocks.** Or, a feature may insert its artifact in an already existing mockup screen. For e.g. a menu item in an existing menu (like adding an option to trigger Mute notifications). Or within a screen. For e.g. checkbox for “No notifications in the night”, which is a widget. In this case, the widget (which will be a E-Block) is owned by the subfeature, and it requests it to be inserted in a mockup of its parent feature - a F-Block owned by parent.
- **Configuration Actions on F-Blocks.** Since F-Blocks allow dynamic changes to themselves as new features are added, we define configurable actions on these blocks. For e.g., a menu F-block will allow any feature to add a word to itself to trigger the intended action. So one of main features will be defining a configurable menu, and later

during development, different features will add their menu entries. The semantics of actions are totally specific to those blocks. The tools used may be able to dynamically create and present the final E-Blocks.
- **D-Blocks** or dynamic blocks. These are in-effect E-Blocks, but dynamically created by running actions on the F-Blocks. They can be recreated by running tools on these specifications.

For the markup, we use independent markup files to capture details of E-Blocks and F-Blocks, and use their reference as part of features and sub-features. D-Blocks are created and stored in directories identifying them as temporary.

Examples of Blocks

Some example of F-Blocks:
- A TopBar of mobile app, which may be configured to contain a menu entry, header, back button, search icon and so on. These are introduced by respective features.
- A dropdown menu, in which different features may add their invocation entrypoints.
- A Screen, whose contents are configured for the main feature that screen implements.

Note that we generate D-Blocks when a feature is planned, and use the screenshots to depict the use of the feature.

Some example of E-Blocks:
- Say, we do “Mute Notifications” popup. It will be designed by UI designer and contain things like duration of notification etc. And thus it will be a E-Block, since we don’t expect any changes by later feature enhancements. If we do make such enhancements, then this will be versioned against older feature, and new version will be associated with new sub feature.

Block Markup Language

We need to keep a list of designed blocks - both E-Blocks and F-Blocks. (G-Blocks are autogenerated.) The spec for the Block is as follows:

```
BlockID: HomeScreenLeftMenu
BlockName: Home Screen Left side menu
BlockType: F-Block
BlockArtifacts:
    ToolType: Vuetify
    File: "HomeScreenLeftMenu.vue"
Attributes:
    MaxSize: 10
    ShowMoreAfterMaxSize: True
```

```
ActionsList:
   - AddMenuWord
       Name: ShortWord
       Position: PositionWord
   - AddToggleWordOption
      Name: ShortWord
      ToggleType: OnOffButton
```

Here, block is logically defined with action names such as “AddMenuWord”. That means, this block can have dynamically added elements from other features. Hence it is F-Block. If it is E-Block, the ActionList will be empty.

Each Action in ActionsList is a name, and it can take one or more arguments as options.

“ShortWord”, “PositionWord”, etc are themselves E-Blocks, supposed to be already in the library.

A Dynamic block is essentially same form as E-Block, except that it will store information about how it was generated.

Associating UI blocks in Feature markup file

The feature will have markup to identify the E-Block (or F-Block with actions and arguments) associated with itself.

```
ID: SearchInGroupChat
...
OwnEBlocks:
   - # None
OwnFBlocks:
   - Id: SearchPopUp  # We can add options, so it is FBlock.
UsingFBlocks:
   - Screen: HomeScreenTopBar   # search is triggered from search icon
there.
     ConfigActions:
        -  Action: EnableSearchIcon
             -  OnClick: SearchPopUp
                SearchIcon: FavIcons.SearchIcon
```

YAML for Managing Code Modules

Like we manage UI mockups with features, we also manage code modules, which act as a definition of what the coders have to implement as part of the feature.

For e.g. if we want “Search” in group chat, then developers may plan to create:

- A generic Search Module, which will be a generic module and allow any schema/contents to be registered and searched.
- And group chat specific search module, which will probably be a plugin, or at least a set of independent functions which:
    - Configures search schema for this feature
    - Subscribes to changes (i.e. new chat etc) as part of group chat, so they are submitted to search engine
    - Implement search API to return results of group chat.

A developer may have to become aware of different side effects, and so on, and that would be documented here as well.

Like in UI artifacts, the actual coding language or methodology doesn’t really matter. We plan to create an intermediate model of Modules which are either generic or specific to features and sub-features. Some examples will help.

Currently our intention is to only create an overall system diagram of modules and their dependencies automatically, and to create a set of their implemented (or about to implement) capabilities for better communication. Later, they would be used to directly map to checkins etc.

Overview of Application Modules

- Modules are also hierarchical i.e. a bigger module may consist of multiple smaller modules.
- Functionality provided by inner modules may be directly exposed or indirectly by container module
- Modules may use other modules through different means:
    - Functional call APIs with well defined semantics
    - REST or other Web Calls with well defined semantics
    - Asynchronously, via message passing
    - Publish Subscribe model - Publishing and/or Subscribing to specific messages
- Modules may own/implement storage objects - Through database calls, File systems or API calls.
- Modules never interact directly with UI elements. However, they may use push messages to update UI elements. UI modules are assumed to be the task of UI subsystem, and hence covered by tools / artifacts within UI subsystem.
- Modules may implement different design patters:
    - Allow plugins to be registered with them (For e.g. multiple auth mechanisms)
    - Be an observer to system events and respond with specific logic

As part of markup, we associate implementation details with features:

- A feature may fully own a module. Or it may reference a module already defined by another feature. (We may have non-UI features also specified in the system to define initial frameworks.)
- When it owns fully, then we may let module's description and specifications fully describe the implementation details.
- When we use existing modules to add features (for e.g. adding URL paths to a web framework), we will use module provided actions. (So a web framework may specify addition of path and associated method/module).

We will provide examples as we define further.

Framework Module (F-Module) Vs. Implementation Module (I-Module)

Any app is built as a set of different modules. A module is supposed to be an independent unit of application, implementing a specific functionality. For example, a Search Module may allow searching different contents used by the application.

A search module is probably a **Framework module**. What it means is that it is designed generally such that other modules may achieve searching by submitting search data to index, and later do a search. We call such modules, Framework Modules. Framework modules should define a set of actions: Usually to register and use the framework in specific manner. The markups should capture such capabilities so that a feature will reference such a module and then register a local module with it (or do other actions).

In contrast, **Implementation Module** will be a standalone module, implementing an API for other modules to use. We capture the intent of such APIs in markup - again using actions. A Feature which has dependency on this module can then provide documentation about how this module will be used.

Example markup of Framework Module, which manages paths in a web server such as python flask:

```
ModuleID: WebAppPathRegistry
ModuleName: This Web app's Path Registration system
ModuleType: F-Module
ModuleArtifacts:
   - Toolkit: Flask
     Path: "./Methods"
ActionsList:
   - AddWebPath
       Path: WebPath
       Methods: WebMethod # GET, POST
      FunctionSpec: FunctionSpec
```

And then, in a feature spec, we will use this F-Block to add a new function, under “Search activity list” in our web app:

```
ID: SearchActivityList
...
OwnIModules:
   - Id: ActivitySearchRegisterFunction
   - Id: SubmitActivityUpdateToSearchFunction
   - Id: SearchActivityFunction
   - Id: HandleSearchFunction
OwnFModules:
   - # None
UsingFModules:
   - Id: WebAppPathRegistry
     Actions:
         -  AddWebPath
               -  Path: “/search”
                  Methods: GET
                  FunctionSpec: HandleSearchFunction
```

Note that our aim is not to write code here! What we want is to capture enough structural details so that the programmer has no questions to ask about what method names to use for this feature. So the check-in system can ensure that only these names are used for implementation, and also implement some semantics around the actions.

YAML for Managing Tests associated with features

This section has not been detailed out. We believe it can be done, and it has many advantages.

Appendix: YAML Specs

YAML schema is specified using the specs here: https://github.com/23andMe/Yamale

References

- Using Stencil component system to generate Ionic components for any specific framework such as React or Vue. https://www.youtube.com/watch?v=RZ6MLELGsD8
- https://plantuml.com/ - Diagrams that we can create using markup
- Feature management tool zepel (https://zepel.io/)

- Diagrams using javascript - can be embedded in wikis: https://mermaid-js.github.io/mermaid/#/
- https://c4model.com/ A lean graphical notation technique for modelling the architecture of software systems. https://www.youtube.com/watch?v=x2-rSnhpw0g shows how to think practically about the architecture diagrams, and not confuse with UML. https://structurizr.com/ is another tool proposed by him, apart from c4 for plantuml.
- https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf - The popular 4+1 model of software visualization
- Many product managers are saying they are using Notion to track product features. Notion is a generic collaboration tool and also a content manager at the same time.