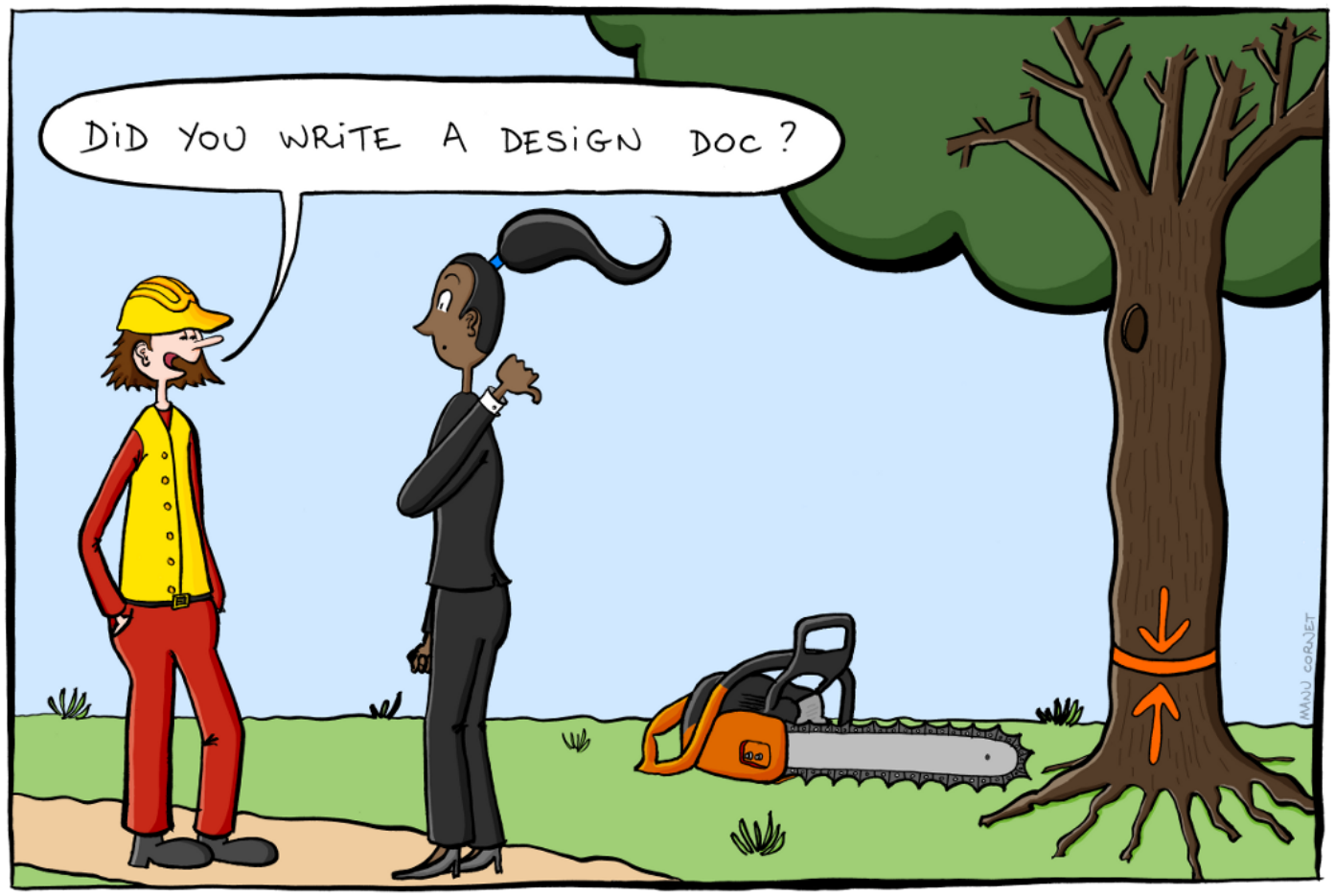
Design Sketch: Animation Subsystem Architecture

Owner: Gio & Sakhya
Last Update: 24 September 2012
Status: draft

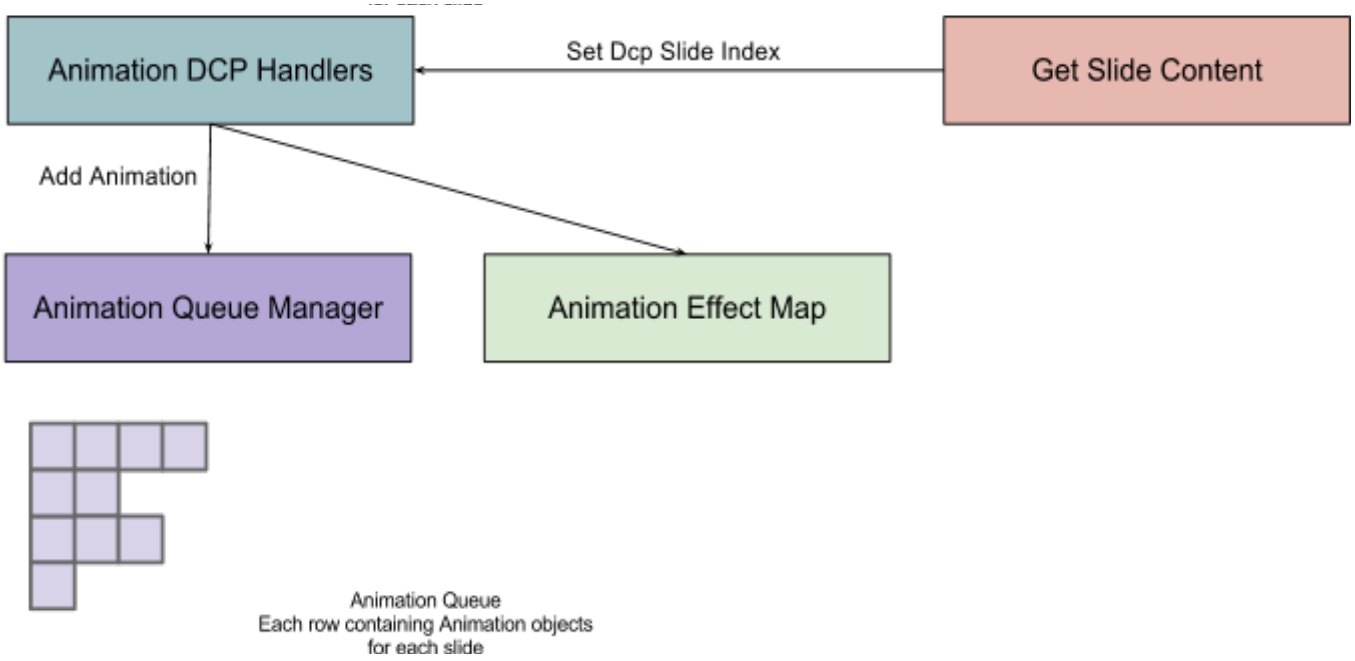
What Design Doc?



https://goomics.googleplex.com/episodeimg/153

QOWT Design Sketch

Animation DCP Handlers



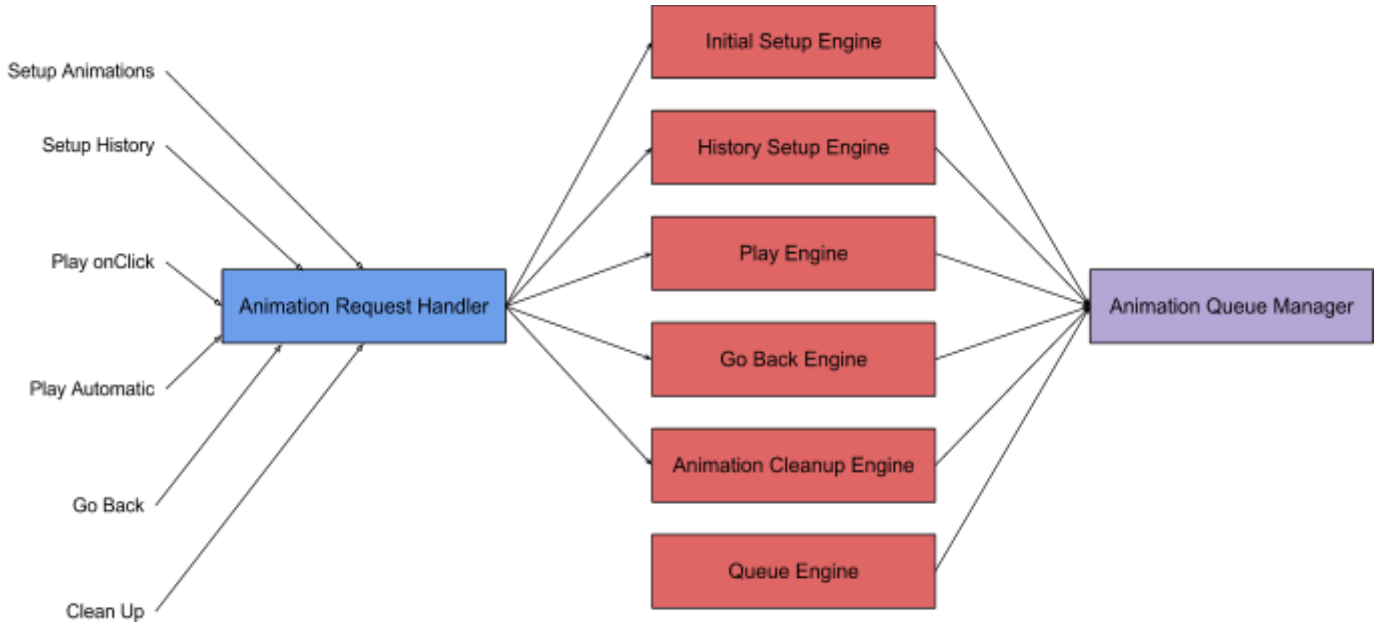
Animation DCP Handlers
Extract animation data for the DCP and create animation objects. The objects are added to the queue in Animation Queue Manager, calling AddAnimation.
There are four Animation DCP Handlers:

1. Timing Handler
2. Condition Handler
3. Common Time Node Handler
4. Shape Target Handler

Get Slide Content Command
Sets the DCP Slide Index needed by the Animation DCP Handlers to know the slide index for each animation.

Animation Effect Map
Used by the Common Time Node Handler, maps the presetID in the common time node to the right animation effect.

Animation Engines



Animation Request Handler
It is the interface with the Point widgets and Presentation Control. Provides an API with methods to set up, play, go back and clean up animations. Depending on the requested action, it talks to the appropriate animation engine.

Initial Setup Engine
It is the engine responsible for the initial setup of the animations, done when moving to the next slide in slideshow mode. It creates an Effect Strategy object for each animation object.

History Setup Engine
It is the engine responsible for setting up the animation history, done when moving to a previous slide in slideshow mode.

Play Engine
It is the engine responsible for playing animations and deals with the different options, for example on Click, With Previous and After Previous animations and animations with Repeat Count. Handles Play onClick, called on click and on key down events and Play Automatic, called when the first animation of a slide needs to be played automatically (if it is a with previous or after previous animation).

Go Back Engine
It is the engine responsible for going back in the animation history. Handles Go Back, called on key down events.

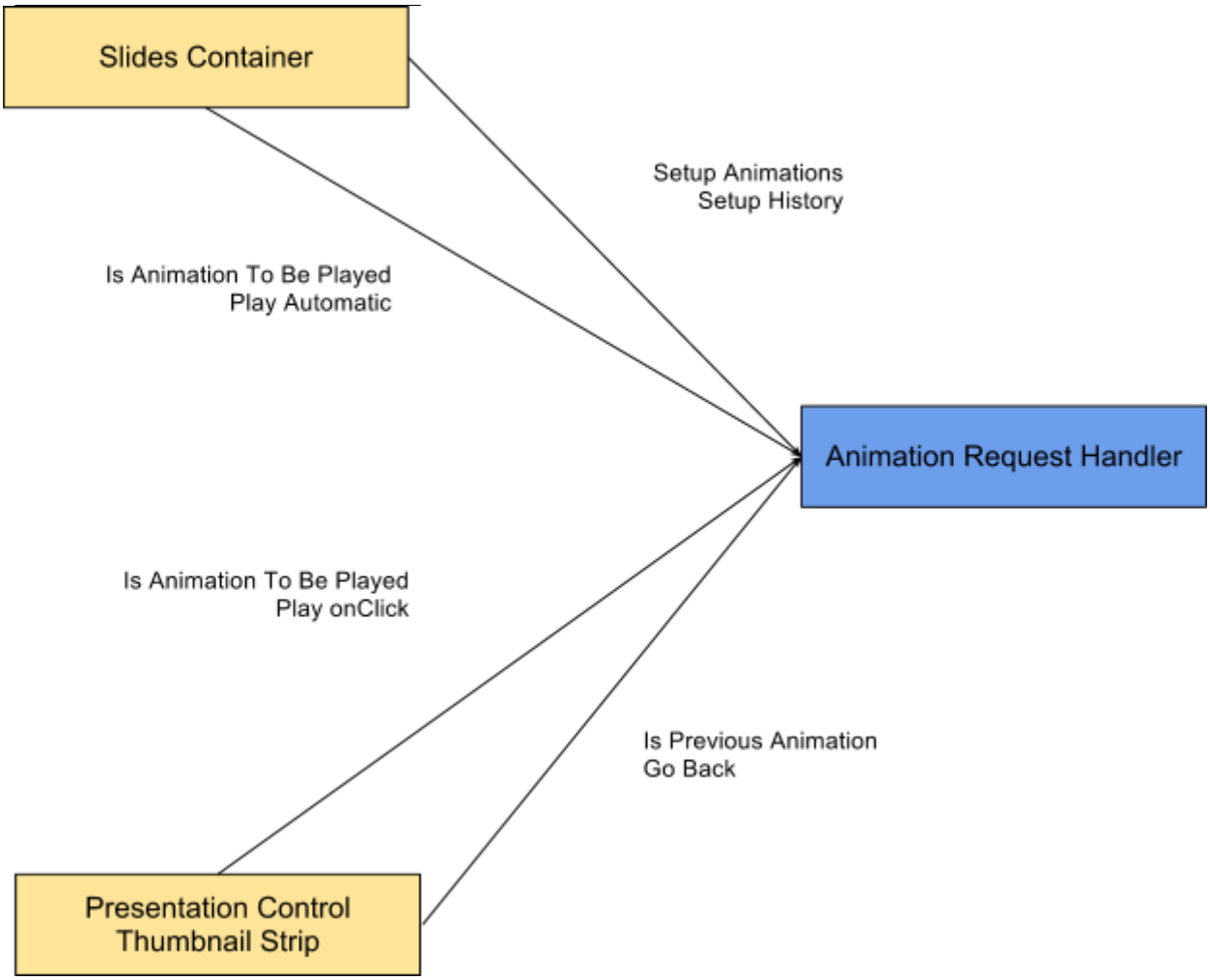
Animation Cleanup Engine
It is the engine responsible for cleaning up the css added for the animations and it is called when exiting the slideshow.

Queue Engine

It provides methods used by the other engines to query the animation queue.

Animation Queue Manager
It implements a queue of animations and it provides an API to access and manipulate the queue.

Clients of Animation Request Handler



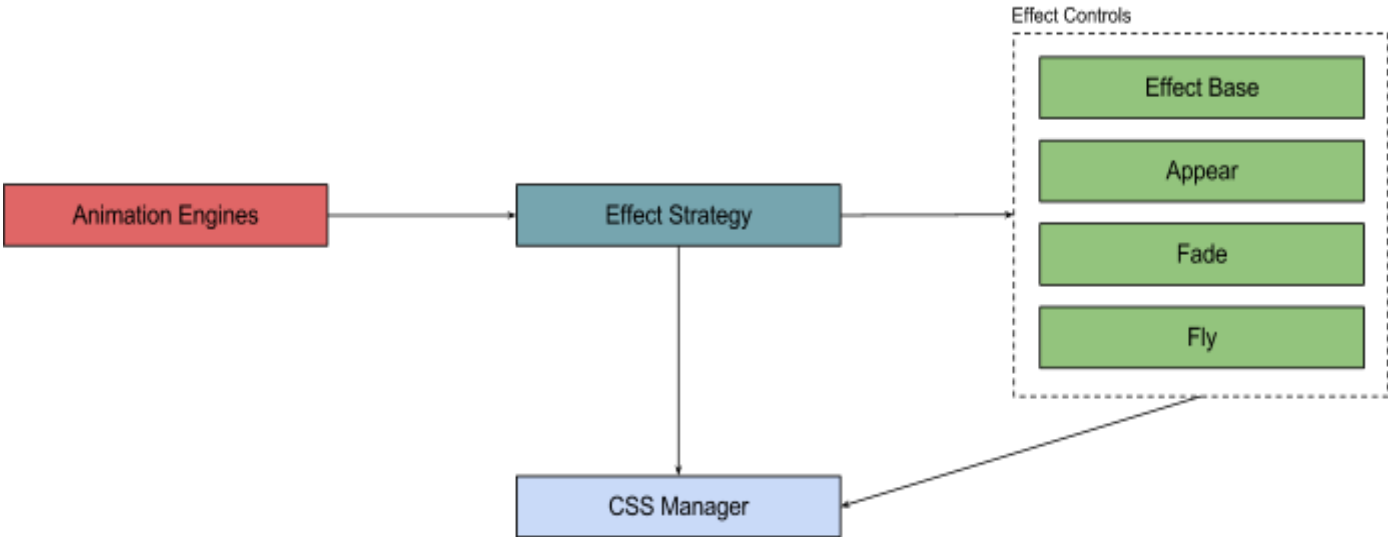
1. Slides Container Widget
    a. When in slideshow mode and moving to the next slide, calls SetupAnimations.
    b. When in slideshow mode and moving to the previous slide, calls SetupAnimationHistory.
    c. When in slideshow mode and the first animation in a slide starts with previous or after previous, calls playAutomatic to make sure that the animation is played as soon as the slide is displayed.

2. Presentation Control
    a. When in slideshow mode it plays animations for an onClick event, calling PlayOnClick. It calls first isAnimationToBePlayed to check that there is an animation to be played, otherwise shows the next slide.

3. Thumbnail Strip Widget

a. When in slideshow mode it plays animations for a key down event, calling PlayOnClick. First it calls isAnimationToBePlayed to check that there is an animation to be played, otherwise shows the next slide.
b. When in slideshow mode it goes back in the animation history for a key down event, calling GoBackInAnimationHistory. First it calls isPreviousAnimationToBePlayed to check that there is an animation before, otherwise shows the previous slide.

Effect Controls



Effect Strategy
An Effect Strategy object is created by the Animation Engine for each Animation object.
It takes care of setting up the initial state of an animation, creating effect objects and setting up the final state of an animation. (called at setUpAnimation)
To set the initial and final state of an animation it uses the addRuleNow method of the CssManager that add rules "just in time".

Effect Base
EffectBase is the base object for all the animation effects.
It defines the interface to add and remove css rules for animations and the effect objects that extend EffectBase can override its methods.
In particular each effect object need to implement the addKeyframesRule method that is specific to each effect.
It provided the implementation of addAnimationRule that add the css for animations using webkit-animation rules and removeAnimationRule.
To handle the CSS, it uses the addRuleNow and the removeRuleNow methods of the CssManager that add rules on the fly. Rules uses as selector the shape id.

Effects
Effect objects (Appear, Fade and Fly) extends the Effect Base and take care of building the CSS animation using -webkit-keyframs rules.

Core/DCP Design Sketch

Core Changes

In the Point KANA core we have made a few changes to support Animations.

The most important being the <fileID, coreID> map. So referring to Animations ECMA spec the animations apply to a particular shape target id. These ID's (lets call it the fileID) are the ones you find it in ECMA spec under shape Target (19.5.72 spTgt (Shape Target)).
However the KANA core generates an unique ID for QOWT to identify each element in the HTML page (and lets call these coreID). In order to apply Animations in QOWT we need to know the unique coreID for the shape target.
This exactly what the <fileID, coreID> map does, it creates a map of the fileID and the coreID, and the DCP merely picks up the unique coreID and passes on to QOWT.

DCP changes

JSON schema

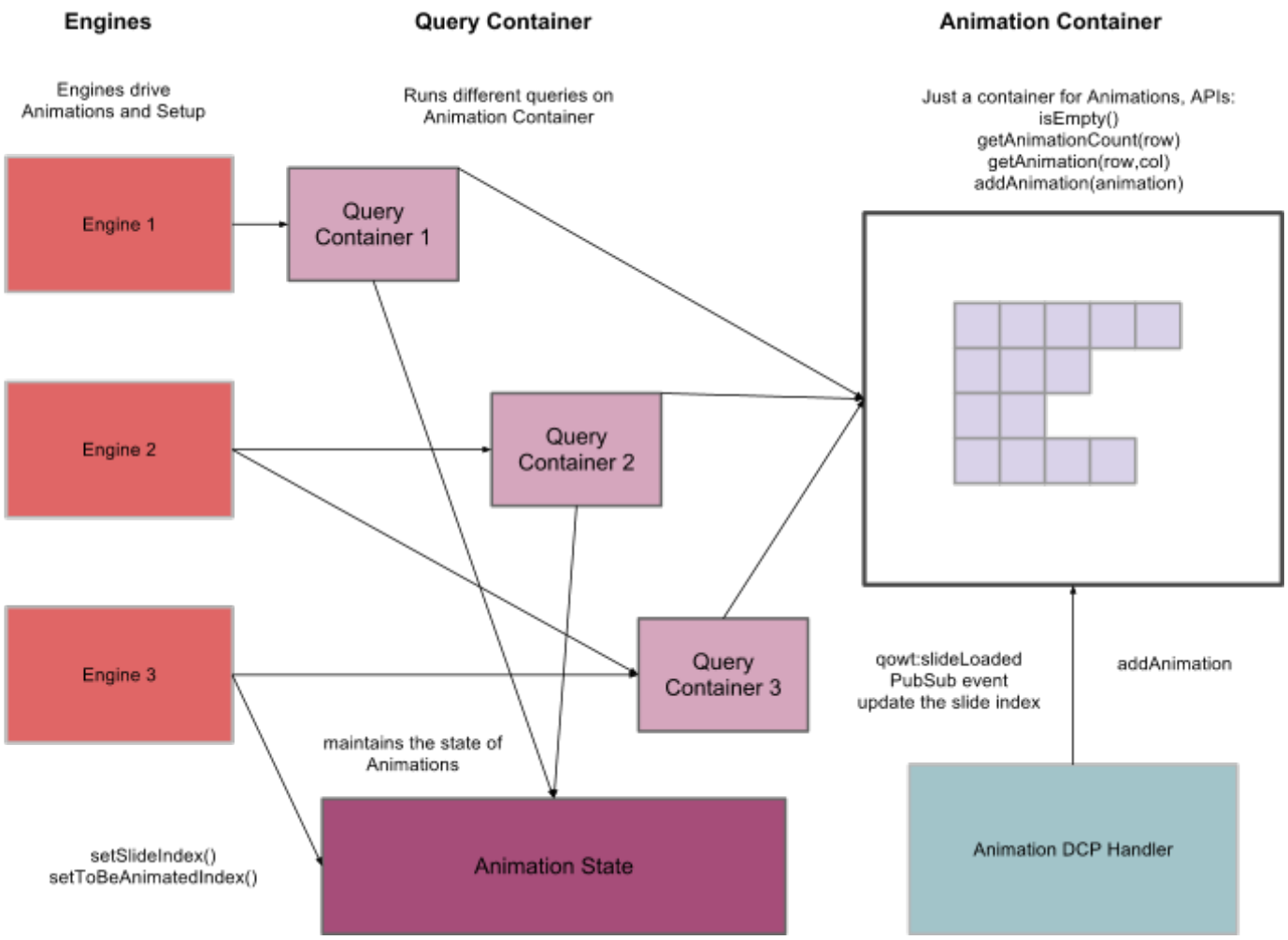
We have nearly followed the ECMA spec whilst creating the JSON schema for DCP (19.5 Animation ECMA-376 Second Edition Part 1).
The reason we have done so is that the Animations are described in SMIL and we thought it would be a good idea to follow the same language to describe animations.
May be in the future if we see that following the complex ECMA spec in our JSON has performance impacts we might want to revisit it.

DCP Visitors

We have a visitor for every JSON schema we have written. We have got a bunch of unit tests as well using the blessed JSON.

Design Improvements

Redesign Animation Queue Manager



The idea is to split the responsibility of Animation Queue Manager into three main components

1. Animation Container
2. Query Container (1,2...N)
3. Animation State

Animation Container

As the name suggests this is the container of Animations. A sort of a matrix, where each row contains Animation objects for each slide.
The API should be redesigned to be one of a container, giving access to the Animation objects for particular row and column.

Sample access API should look like:
`isEmpty(), getAnimationCount(), getAnimation(row,col)....`

The DCP handlers are responsible for adding the animations into the Animation Container (calling addAnimation(animation)).
The Animation Container should now listen to qowt:slideLoaded PubSub event to update the slide index (to know which row/slide the new animations are to be added).

Query Container

The responsibility of the Query Container is to run various complex queries on the Animation Container. All access to the Animation Container happens through the Query Container.
The engines are responsible for setting up the animations and playing the animations. However the engines from time to time need to run complex queries on the Animation Container. The idea is to create a bunch of Query Containers that will serve the Engines. In this way we are deliberately keeping the Animation Container simple and having a container-like interface. The Query Container will now be responsible to access the Animation Container and interpret the data.

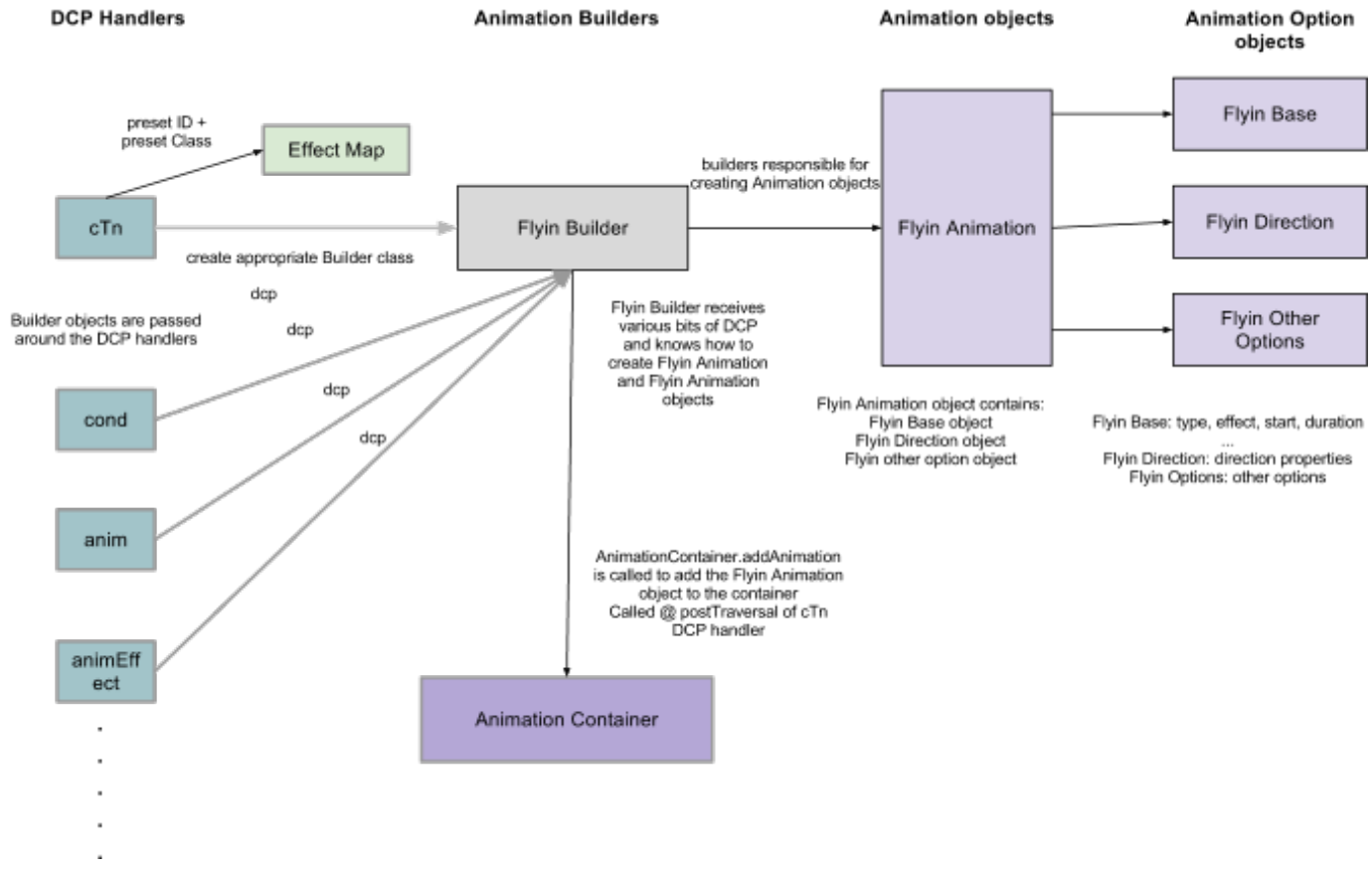
Sample API should look like:
`isAnimationToBePlayed(), getPreviousAnimation()`

Animation State

The responsibility of this module is to maintain the animation state on behalf of the Animation Container.
As the Engines drive the animation and gets access to the Animation Container via the various Query Containers the slideIndex and the toBeAnimatedIndex is maintained by the Animation State module.
The Engines will manipulate the slideIndex and the toBeAnimatedIndex via the Animation State module API interface. The Query Containers will in turn retrieve the state (indexes) from the Animation State module.

Sample API should look like:
`setSlideIndex(), setToBeAnimatedIndex(), getSlideIndex(), setSlideIndex()`

Redesign the interaction between DCP Handlers and Animation objects



Animation Object

The purpose of this redesign is that as we add new Animation options, various types of Animation objects will need to contain a lot more information/properties.
The new design aims to manage the fact that different Animation objects will have various options (in the diagram above Flyin Animation is used as an example showing that Flyin Animation should in the future have Flyin Directions and various other options)

Every Animation object should be contain the following objects:

1. Animation Base object (mandatory)
2. One or more Animation Option objects (optional)

Animation Base

Every Animation object should contain the Animation Base object. This as the name suggests should have various bits of information that are common to all Animation objects.
`(Animation Type, Animation Effect, Animation Start, Duration, Delay, Repeat Count)`

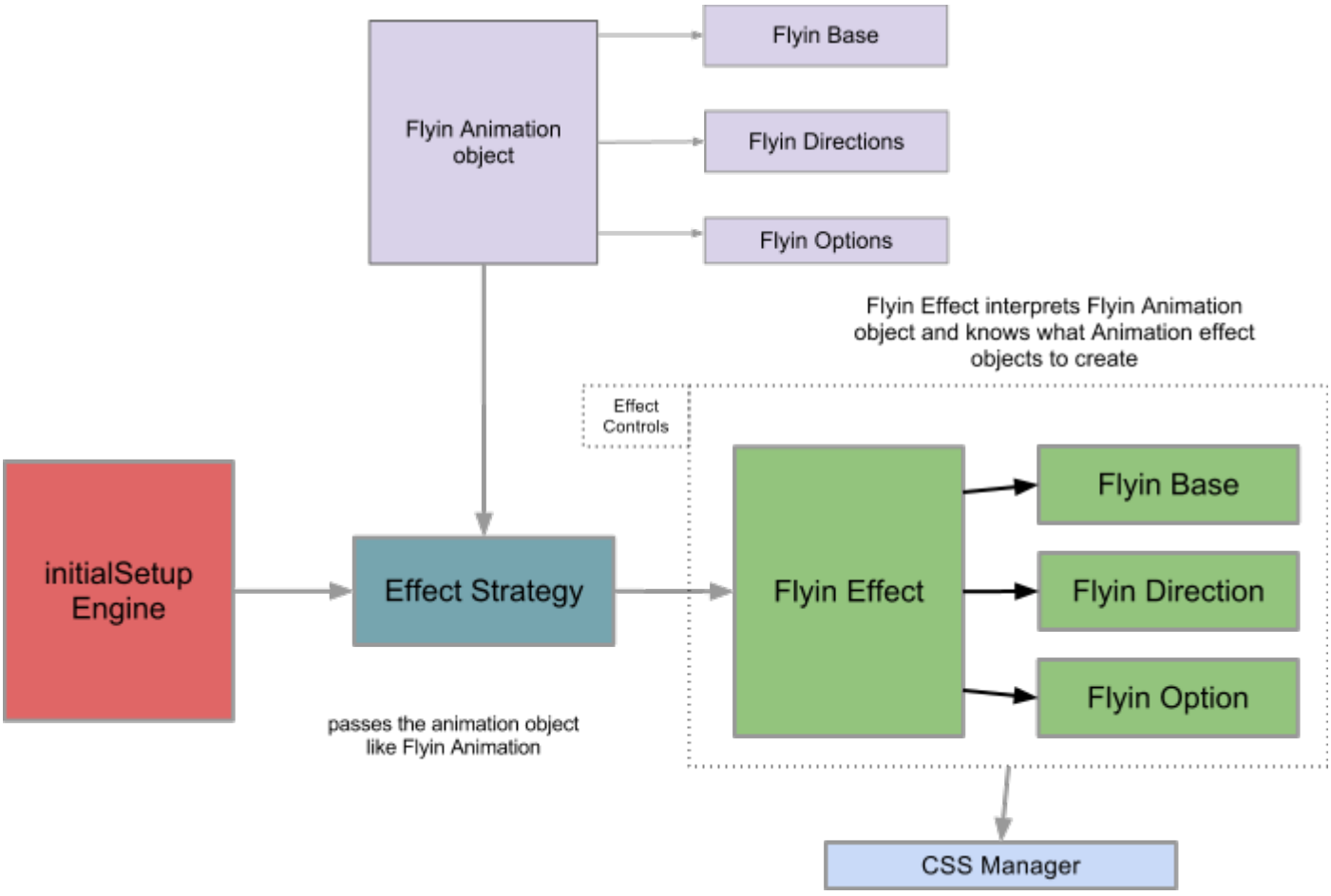
Animation Option

The Animation object could have one or more Animation Option objects. These objects will encapsulate the various Animation options.
As we support various complex animations the Animation objects should contain various Animation option objects.

Animation Builder

As you have various types of Animation objects you would need various Animation Builders that knows precisely how to build these Animation objects.
The DCP handlers would pass the Animation Builder objects among each other and feed the Animation Builder with the dcp data and the Animation Builder would know how to build the appropriate Animation object.
The CommonTimeNode DCP Handler liases with Animation Effect Map (should feed both preset ID and preset class) to understand the type of the Animation Builder it has to create. (might be a good idea to use a factory to create various Builders).
Once the animation type is known with the Animation Effect Map, the appropriate Animation Builder object is created by the CommonTimeNode DCP Handler. The Animation Builder object is then fed with the dcp and the Animation Builder knows how to create the appropriate Animation object and their Animation option objects. The Animation Builder is passed around various DCP Handlers and they in turn feed the Animation Builders with various DCP data.
At the post traversal of the CommonTimeNode DCP Handler, the Animation object should be fully constructed, and hence it would be a good time to add the Animation object to the Animation Container.

Redesign Effects and Effect Strategy



The purpose of the redesign of the Effect Strategy and the Effects is similar to the one for Animation objects. As various types of Animation objects will need to contain a lot more information/properties we need to support different Effects for each of those Animation options. Thus the Effects now follow the same object layout of the Animation object.

Effect Strategy

An Effect Strategy object is created by the Initial Setup Engine and attached to each Animation object. An Effect Strategy object knows which Effect object need to create for each animation. Now an Effect Strategy also sets the CSS for the initial and final states of an animation. The idea is to move all the logic that handles the CSS in the Effect object and Effect Options objects and keep the Effect Strategy as a simple API that provide the bridge between the Engines and Effects. Each method in the API should just call the corresponding method in the Effect object.

Sample API should look like:
`play(), setupInitialState(), setupFinalState()`

Effect object

Every Effect object should contain the following objects:

1. Effect Base object (mandatory)
2. One or more Effect Option objects (optional)

Effect Base

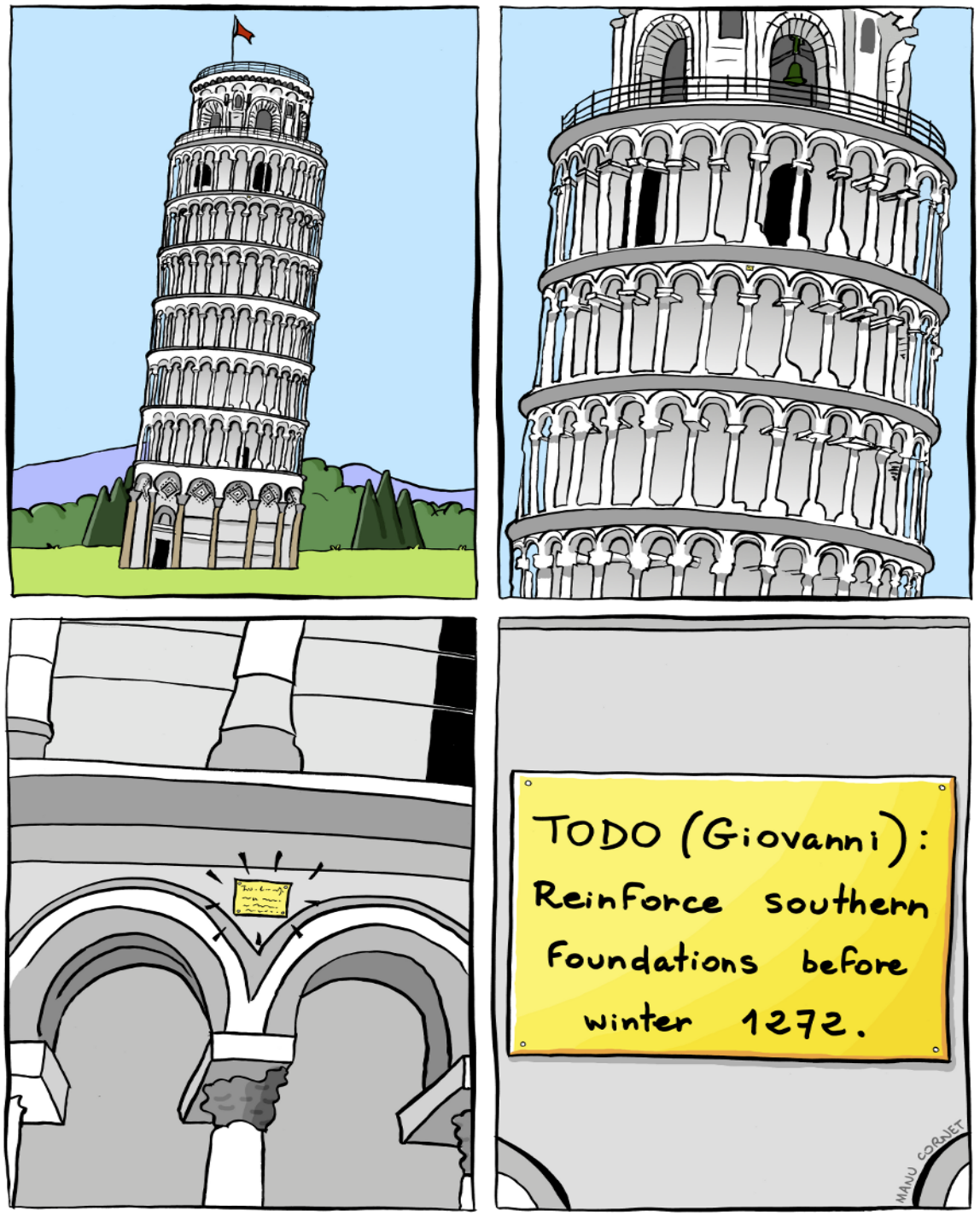
Every Effect Object should contain the corresponding Effect Base object. This as the name suggests should create all the CSS rules for an animation that are common to that specific effect. In the example diagram a FlyInBase would create the css rules that are common to all the FlyIn effects.

Sample API should look like:
`addKeyframesRule(), addAnimationRule(), removeAnimationRule()`

Effect Option

The Effect object could have one or more Effect Option objects. These objects will encapsulate the various Effect options.
In the example diagram above a Flyin Effect will contains the Flyin Base Effect and Flyin Direction Effect and possibly some other Flyin options.

TODO() { TODO(); }



https://goomics.googleplex.com/152

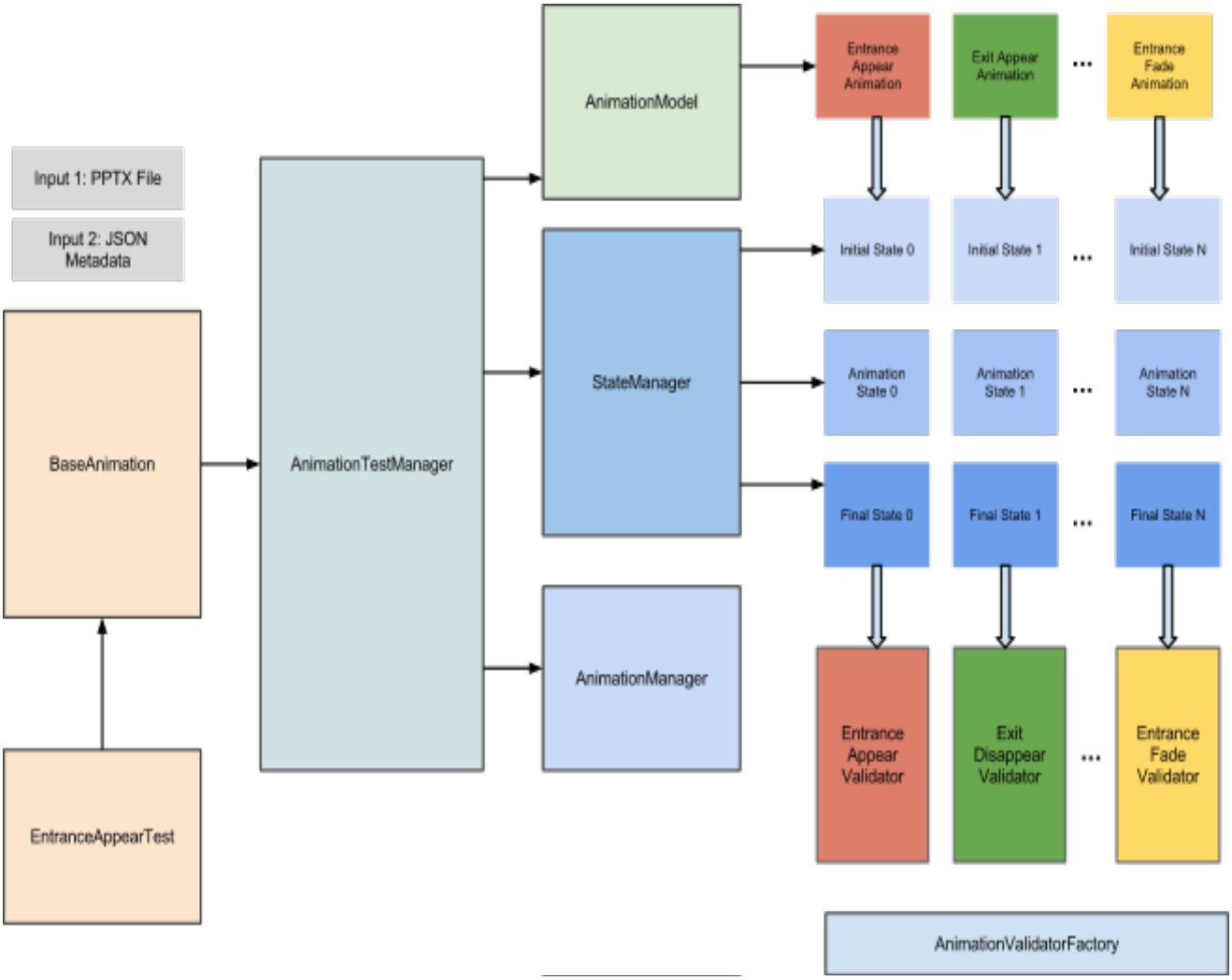
We are planning to put down the list of TODO items in a priority order:

1. Update Animation Effect Map. Create a map of presetID and presetClass to correctly identify the animation type.
2. Redesign Animation Queue Manager (see the diagram above).
3. Redesign the interaction between DCP Handlers and Animation objects (see diagram abovr).
4. Remove the EffectBase and use composition for the effects (FlyIn Base, FlyIn Direction, etc) (see the diagram above)
5. We need to have unique IDs for a slide thumbnail and the slide itself, hence need an efficient mechanism to create and assign ids to elements in the slide so we can apply animations to it.
6. Remove Animation.create() in Timing Handler and check in the other handlers that the object has been created, otherwise don't do anything. (Bug should be fixed)
7. Use "qowt:slideLoaded" PubSub signals in Get Slide Content and remove preExecuteHook(). Change Animation Queue Manager to listen to that signal and update slide index.
8. Write more QOWT unit tests, in particular to test the engines.
9. In EffectStrategy when we add support for emphasis animations remove EmphasisEffect, that is just a placeholder.
10. Add more Automated tests to support (WithPrevious/AfterPrevious, Repeat Count, Duration, all features that we are going to support)
11. Migrate the clients of Animation Request Handler to use Tools, contentManager and "do:action" signals.

Tests

Automated Test

Design Sketch



Objective

The mission is to make the automated tests for animations scalable, reusable and easy to write.

The reason writing tests for animations is difficult is because the number of variable parameters:

1) Multiple Animation Effects (Entrance/Exit/Emphasis -> Appear/Fade/Flyin..)
2) When animation starts (onClick, withPrevious, afterPrevious)
3) Multiple animation Options (Delay/Duration/Repeat...)
4) Animation Specific Options (Flyin Directions...)
5) Animations on multiple shapes (Shape, Tables, Images...)

Approach

We have designed the automated tests so that we solve the above mentioned problems.

Input JSON metadata
To tackle the problem of so many variables we have introduced a new mechanism to describe data, meta data. The meta data is one of the inputs alongside the ppt/pptx file. The JSOn metadata explains the animations in the pptx file. So it is flexible and can be changed as the input file pptx changes (without any hard coded values in test files.)
The format of the data is very similar to the JSON animation objects stored in QOWT. We use the Google GSON libraries to convert the JSON objects into JAVA objects.

Validator objects
The way we have done it is to refactor the code so that the validation happens through individual validators created on the fly and can be easily reused. The tests run through the JSON metadata and on the fly creates these validator objects based on the type of animation. For example if it is a Entrance Appear animation, the validation would be delegated to the Entrance Appear Validator which knows how to validate the particular effect.

Three States - Initial / Animation / Final State
The idea here is that at the start of every test class, we read the JSON metadata, which explains the entire animation sequence on the shapes.
First we save the initial states of all the web elements participating in the animation. By states we mean all the important CSS properties that exciting (visibility, animation-name, animation-mode, animation-duration..).
Then we then run through the animation sequence as described in the JSON metadata, and save the state of the animating element. We call this the animation state. We continue doing for each animating element one after the other.
Finally we save the final state of all shapes. All the states are managed by the StateManager and the animation sequence is ran by the Animation Manager.

Easy to write tests
The tests that we write have been intentionally kept very simple and generic. The good thing about it is as we have kept it generic it is easy to add more and more tests with various animation effects and with various animation options on various shapes.

Unit Test

At the moment we have unit tests in the following components:

1. DCP Handlers: we have a few unit tests for some of the DCP Handlers. In future we should have unit tests for each DCP handler
2. We have some tests for Animation Container (or the Animation Queue Manager), but after some analysis we think it is not be efficient to write more unit tests for the Animation Container

Unit tests Improvements

1. We should add unit tests for the Engines, as they are heart of the Animation sub system, so the idea should be we write on unit test for each Engine
2. Optional unit tests for Animation Request Handler