

Rapport de stage
Programmation Enyo en OCaml

Vincent Botbol

17 août 2012

Table des matières

1	Introduction	3
2	Présentation Enyo	5
2.1	Introduction	5
2.2	Débuter en Enyo	6
2.3	Ajout de sous-composants	7
2.4	Événements	8
2.5	Propriétés published	9
2.6	Copie, partage et réutilisation de composants	9
2.7	Hierarchie des objets	10
2.8	Bibliothèques additionnelles	11
2.8.1	Onyx	12
2.8.2	Canvas	12
2.8.3	Layout	12
3	Interfaçage d’Enyo en OCaml	13
3.1	Représentation des objets	14
3.1.1	Modélisation	14
3.1.2	Création	15
3.1.3	Instanciation	16
3.2	Gestion des propriétés et des méthodes	17
3.3	Gestion des événements	19
3.3.1	Fonctionnement	19
3.3.2	Introspection	22
3.3.3	Autre stratégie de propagation	22
4	Langage de Description d’Interface (IDL)	25
4.1	Interface des objets	25
4.2	Générateur de code	27
5	Conclusion	28

6	Annexes	30
6.1	Hiérarchie complète Enyo	30
6.2	Déploiement d'applications	32
6.3	Exemples de programme	32

Chapitre 1

Introduction

En quelques années, les appareils mobiles se sont peu à peu imposés sur le marché technologique. Grâce à leurs interfaces graphiques simplifiées et leurs fonctionnalités, ces outils ont su se rendre indispensable auprès du grand-public. Aujourd’hui, environ un tiers de la population possède un smartphone ou une tablette.

La variété de ces appareils est telle que la tâche est parfois agaçante pour les développeurs d’exporter leurs applications sur d’autres systèmes utilisant des langages différents. La nécessité d’une unification s’est donc imposée. Pour se faire, le choix des constructeurs quant à la technique s’est dirigé vers les nouvelles technologies web : HTML5 et JavaScript. Populaire et en constante évolution, ces deux solutions apportent un grand nombre de possibilités mais possèdent néanmoins leurs désavantages.

Le but de ce stage est de permettre au programmeur de développer des applications portables de manière sûre en passant pour cela par le langage OCaml garant d’une solidité à travers la puissance de son typage statique tout en gardant le confort de programmation qu’on lui connaît. Pour se faire, on s’est intéressé à la bibliothèque JavaScript “Enyo” (2) en interfaçant celle-ci afin de satisfaire notre objectif.

La traduction d’OCaml vers JavaScript a déjà été mise-en-oeuvre dans plusieurs travaux : O’Browser et Js_of_ocaml. Fournissant des bases intéressantes pour ce projet, il a été pratique d’en faire usage pour s’abstraire du problème d’inter-opérabilité entre les deux langages.

La divergence entre JavaScript et OCaml est importante. Cela peut beaucoup jouer lors d’un tel interfaçage. Il est alors important de bien étudier la bibliothèque afin de déterminer le niveau d’utilisation des traits intrusifs. En répondant cette question, il faut pouvoir déterminer le type de modèle et sa représentation s’adaptant le mieux. Les objets Enyo étant nombreux, il peut être important de se donner les moyens de les représenter de façon efficace. Se munir d’un IDL peut-il être un atout pour réaliser ceci ?

Premièrement, nous étudierons Enyo pour nous permettre d’obtenir les

informations nécessaires pour établir une approche.

Ensuite, nous tenterons de fournir un modèle en détaillant les raisons de ce choix et les problèmes qui en ressortent.

Enfin, nous présenterons un IDL permettant de modéliser les objets Enyo.

Chapitre 2

Présentation Enyo

2.1 Introduction

Enyo[4] est un framework Javascript orienté-objet. Il privilégie un style de développement basé sur une encapsulation arborescente de composants.

Dans sa première version, Enyo fut employé pour le développement d'application des tablettes HP TouchPad fonctionnant sous webOS. Après un succès mitigé, HP décida de transformer webOS en projet *open source* devenant ainsi : Open WebOS.

Enyo passe alors en version 2.0. Sa conception en Javascript profite de l'engouement actuel pour ce langage et permet aux développeurs de déployer leurs applications sur les plateformes le supportant, ce qui aujourd'hui désigne la plupart des appareils grand public (Android, iOS, WindowsPhone, etc.)

On peut attribuer à Enyo des avantages tels que sa prise en main aisée grâce au style de programmation proche d'une vision du DOM, le poids relativement faible de sa bibliothèque et sa possibilité d'extension relativement aisée.

2.2 Débuter en Enyo

Il est important de distinguer deux phases dans le développement d'une application Enyo.

- La première, dite de création, consiste à définir les composants, leurs positions dans la hiérarchie de l'application ainsi que le modèle de calcul.
- La seconde est celle du déploiement. Permettant à la bibliothèque Enyo de générer, à partir du squelette décrit par la phase de création, l'objet destiné à être instancier puis afficher.

La définition des composants de l'application consiste, vulgairement, à déclarer des objets Javascript possédant des propriétés qui auront un sens au moment d'étendre l'application.

Par exemple, définissons un simple morceau de texte contenant la chaîne "Bonjour monde!" :

```
1  var monApplication = {kind: "Control",
2                          name: "MonApp",
3                          content: "Bonjour monde!"}
```

Ici, nous avons simplement créé un objet Javascript muni de propriétés suffisantes pour renseigner Enyo sur le type d'objet à créer.

Explicitons ces propriétés :

- *kind* définit le type d'objet Enyo que nous souhaitons créer.
- *name* nomme l'objet que nous sommes en train de créer. C'est par la valeur de cette propriété que nous pourrions accéder à un composant ou encore l'instancier.¹
- *content* attribue un texte au composant. Ceci n'a de sens que parce que l'on manipule un objet Enyo de type "Control" ou un de ses descendants qui possède donc cette propriété.

Désormais défini, il faut passer le composant à la méthode *kind* d'Enyo pour finaliser l'application et la rendre instanciable.

```
1  enyo.kind(monApplication);
```

A la suite de cet appel, on pourra constater qu'une nouvelle classe *MonApp* est présente dans l'environnement. Il ne reste donc plus qu'à l'instancier pour pouvoir ensuite l'afficher.

1. Il est important de noter que si placée dans un composant en racine de l'application, la propriété *name* déterminera le nom de la classe générée. Autrement, le nom d'un sous-composant servira de clé à une table d'association définie sur le composant racine de l'application servant à récupérer les divers sous-composants

```

1  var app = new MonApp();
2  app.renderInto(document.body);

```

En s'intéressant de plus près à l'objet "app", on peut constater qu'il possède un nombre important de méthodes et de propriétés que nous n'avons pas forcément définies à la création, telle que la méthode *renderInto* que nous venons d'utiliser.

Ces méthodes ont été ajoutées par induction lors de la construction de l'application par la méthode *enyo.kind*.

Si le champ *kind* spécifie un "Control", il est donc nécessaire à Enyo de lui rajouter toutes les propriétés et méthodes que possède un Control Enyo qui n'ont pas été définies par l'utilisateur.

2.3 Ajout de sous-composants

Enyo suivant une logique d'arborescence de composants. Lorsque l'on crée son application, il est nécessaire de pouvoir lui ajouter des sous-composants. On réalise cela en définissant la propriété *components* avec en valeur le tableau des composants que nous souhaitons.

Imaginons ici que l'on souhaite créer une application consistant d'un champ de saisie et d'un bouton de validation :

```

1  enyo.kind({kind: "Control",
2             name: "MonApp",
3             components: [
4                 {kind: "Input"},
5                 {kind: "Button", content: "ok"}
6             ]
7             });

```

La fonction *kind* va récursivement créer tous les sous-composants de la hiérarchie ainsi que leur méthodes associées.

Comme précédemment, pour l'afficher, on instancie la nouvelle classe ainsi générée et on lui applique une méthode d'affichage :

```

1  var app = new MonCadre();
2  app.renderInto(document.body);

```

Si l'on observe le code html après ces appels, on s'aperçoit qu'enyô a injecté dans le DOM une balise html de type <div> dans laquelle sera placée, respectivement, deux balises : un <input type="text" ..> et un <button>.

Notons que nous n'avons pas défini la propriété *name* des sous-composants. Rappelons que le nom d'un composant sert uniquement à l'accession de celui-ci par les autres composants de la hiérarchie ou encore pour son instantiation. Cependant, lors du *kind*, seule la racine de l'arbre sera étendue à l'environnement.

Il faut aussi noter que si la propriété *name* n'est pas définie, Enyo s'en charge en lui donnant un label généré ("input0" et "button0" ici). Il n'est évidemment pas recommandé d'utiliser ces derniers pour y accéder.

2.4 Événements

Reprenons l'exemple précédent et supposons désormais que l'on souhaite, lorsque l'on clique sur le bouton, afficher un message d'alerte.

```
1 enyo.kind({kind: "Control",
2           name: "MonCadre",
3           components: [
4             {kind: "Input", value: ""},
5             {kind: "Button", content: "ok",
6               handlers: {ontap: "monCallback"},
7               monCallback: function(sender, event){
8                 alert("ok");
9                 return true;
10              }
11           ]
12         });
```

Au bouton, nous avons rajouter une propriété *handlers*. Cet objet sert à définir tous les événements que l'objet va traiter. Cela peut-être des événements du dom (onkeyup, onload, ...) comme des événements spécifiques à la plateforme sur laquelle l'application est déployée (l'inclinaison d'une tablette par exemple). Ici, nous utilisons l'événement "ontap", alias d'"onclick".

A chaque propriété-événement est attaché une chaîne contenant le nom de la fonction à appeler.

Nous définissons donc la fonction click qui prend en premier argument l'émetteur de l'événement et l'événement envoyé contenant ses propriétés.

La valeur de retour de cette fonction spécifie si l'événement doit s'arrêter ou remonter ("bubble") aux composants parents. On retourne vrai pour l'empêcher de remonter et faux pour le laisser "bullé" sur ses ancêtres.

On notera qu'en général, les fonctions de traitement ne pas censées devoir "buller" sont définies à la racine de l'arbre qui possède un accès à tous ses sous-composants grâce à sa table de hash "\$".

Il existe quelques spécificités pour la manipulation d'événements :

- Si la fonction de traitement n'est pas trouvée, l'événement est remonté dans l'arbre de composants jusqu'à trouver une fonction dont le nom correspond.
- Il n'est nécessaire de déclarer le bloc *handlers* que lorsque la fonction de traitement se situe dans le même objet que celui-ci. Par exemple, cette définition est parfaitement valable :

```
1 enyo.kind({kind: "Control",
2           components: [{kind: "Bouton",
3                         ontap: "traiteClique"}
4                     ],
5           traiteClique: function(sender, event){
6             alert("Click reçu"); return true
```

```

7         }
8     });

```

- Il est possible de propager un événement manuellement en utilisant la fonction `send(nomEvenement, evenement)` de l'objet `enyo.Signals`. Cet événement est remis à tous les objets de type `kind : "Signals"`. Il suffit alors à l'objet-récepteur de posséder un sous-composant de ce type et d'implémenter la fonction de traitement lié au handler du sous-composant. (voir exemple)

2.5 Propriétés published

Les propriétés placées dans l'objet "published" du composant sont en général utilisées pour la logique de l'application. Certains "published" sont déjà présents dans les objets Enyo. Par exemple, la propriété "content" de l'objet "Control" est elle-même publiée.

Enyo traite ces propriétés comme étant des variables publiques et génère automatiquement : getter, setter ainsi qu'une méthode `valueChanged` appelée à chaque appel du set avec une nouvelle valeur.

En résumé, un composant déclaré ainsi :

```

1 {kind: "app", published: {maVariable: 0}}

```

est équivalent à :

```

1 {kind: "app", maVariable: 0,
2   getMaVariable: function() {..},
3   setMaVariable: function(nouvelleValeur) {..},
4   maVariableChanged: function(ancienneValeur) {}}

```

Cependant il reste possible d'initialiser les valeurs de propriétés déjà publiées hors de l'objet *published* de la même façon dont nous procédions avec le champ *content* jusqu'à présent. Cela aura uniquement pour effet de masquer la première initialisation que le constructeur Enyo aura effectué.

Note : la propriété `monObjet.published.maVariable` ne sert qu'à indiquer au constructeur d'objet Enyo qu'il doit générer la variable et sa barrière d'abstraction. Des effets de bords sur cette variable après instanciation n'auront aucun effet.

2.6 Copie, partage et réutilisation de composants

Il est possible de réutiliser des composants déjà créés. Il faut cependant faire attention à ne pas lui attribuer de nom. Dans le cas échéant, le constructeur considérera que l'objet existe déjà et tous les traitements visant une des occurrences de ce nom seront appliqués au premier objet défini.

```

1 var unBouton = {kind:"Button", name:"bouton1",
2                 handlers:{ontap:"clique"},
3                 content:"un bouton",
4                 clique:function(){
5                     this.setContent("Reception du click")
6                 }
7 };
8
9 enyo.kind({name:"App", components:[unBouton,unBouton]});

```

Dans cet exemple, au moment de l’instanciation un message d’avertissement concernant une collision de nom est émis mais les deux boutons s’afficheront correctement. Cependant, on constate bien que les clicks émis sur l’un ou l’autre des boutons n’auront pour effet de changer que le premier des deux boutons.

Typiquement, si l’on souhaite réutiliser un composant déjà implémenté, il est recommandé de procéder ainsi :

```

1 enyo.kind({kind:"Button", name:"MonBouton",
2           handlers:{ontap:"clique"},
3           content:"un bouton",
4           clique:function(){
5               this.setContent("Reception du click")
6           }
7 });
8
9 enyo.kind({name:"App",
10           components:[
11               {kind:"MonBouton", name:"bouton1"},
12               {kind:"MonBouton", name:"bouton2"}
13           ]
14 });

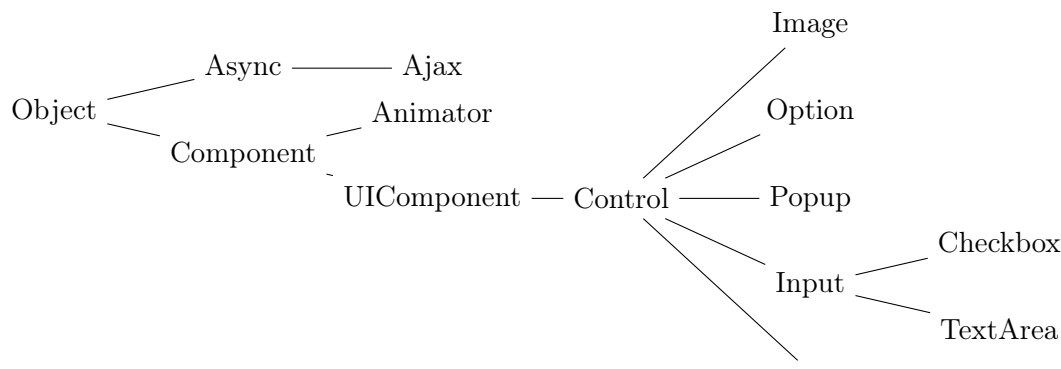
```

Une nouvelle classe *MonBouton* est alors étendu à l’environnement qui pourra être “Kindé” comme un objet Enyo standard dans une définition de composant.

2.7 Hiérarchie des objets

On peut considérer la plupart objets Enyo dans la hiérarchie comme des noeuds HTML, en effet, lorsque l’on construit son application, la forme arborescente de l’arbre se traduit particulièrement bien sous une forme HTML. Ainsi, un *Button* Enyo est lui-même transformé en un noeud “<button>” à l’instanciation. Il est donc aisé de se représenter le rendu final d’une application.

Voici un aperçu de l’arbre des composants d’Enyo :



L'arbre complet peut-être trouvé en annexe (6.1).

La plupart des objets héritent de la classe *Control*. En effet, c'est à partir de celle-ci que l'on obtient des fonctions d'affichage et la génération d'un noeud html. Les objets parents de *Control*, tel que *Component* ou *Object* ne servent, en pratique, qu'à établir une logique dans le code (création de temporisateurs, calculs de valeurs, etc.). On peut donc tout à fait se passer de ceux-ci lors de l'utilisation en OCaml².

2.8 Bibliothèques additionnelles

Enyo possède un ensemble de bibliothèques supplémentaire apportant du contenu à l'API de base. On peut par exemple utiliser la bibliothèque Onyx pour fournir une interface graphique supérieure ou encore Layout pour organiser ses composants de la manière souhaitée.

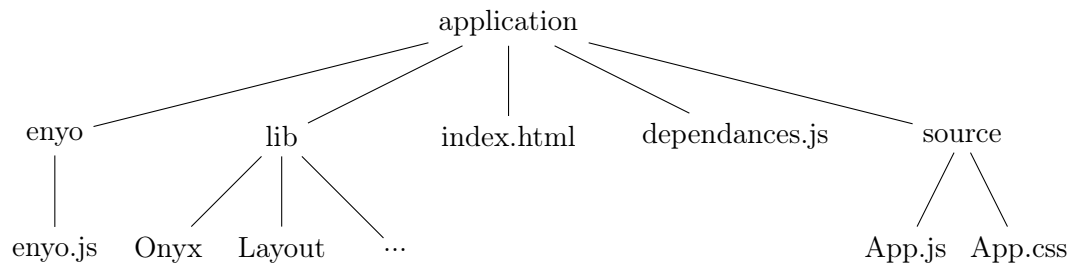
Ces objets viennent généralement se greffer à la hiérarchie existante. Par exemple, "Onyx.Input" hérite de "Control".

Pour pouvoir utiliser cette extension, il est nécessaire de l'indiquer à l'application. Il faut faire appel à *enyo.depends* sur la liste des bibliothèques à lier. Par exemple, si l'on souhaite utiliser des objets d'"Onyx" et de "Layout", il faut procéder ainsi :

```
1  enyo.depends{ "$lib/Onyx", "$lib/Layout" }
```

Il est bien sûr requis de fournir le dossier contenant ces extensions au même niveau que le dossier "enyo". Voici une arborescence classique d'une application Enyo :

2. La logique de l'application étant gérée par le programme OCaml



2.8.1 Onyx

La bibliothèque Onyx enrichit l'application en apportant un ensemble de style prédéfinis pour des objets de bases. L'objectif étant d'obtenir une interface graphique correcte sans avoir à passer du temps sur le style de l'application.

2.8.2 Canvas

Cette extension gère la partie canvas d'HTML5 d'Enyo. Elle définit des objets pour simplifier le code JavaScript et ainsi s'abstraire légèrement de la couche basse d'HTML5.

Cependant, cette bibliothèque a ses limites et l'intérêt décroît à mesure que l'application se complexifie. Il est parfois inévitable de recourir aux fonctions de bases de manipulation de canvas pour obtenir le résultat escompté.

2.8.3 Layout

Permet de disposer les composants de l'application avec flexibilité. L'usage de cette bibliothèque n'est malheureusement pas très intuitive, elle nécessite exemples et lecture de la documentation présente sur le site d'Enyo.

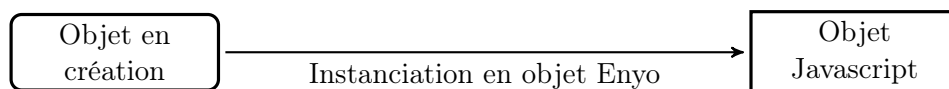
Chapitre 3

Interfaçage d'Enyo en OCaml

Pour une adaptation en OCaml d'Enyo, il est nécessaire d'établir une communication entre les deux langages. Cela présente une première difficulté étant donné le peu de points communs existants. Nous pouvons néanmoins résoudre le problème en utilisant *Js_of_ocaml*, un compilateur de byte-code OCaml vers Javascript développé par l'équipe d'*Ocsigen*. Nous obtenons ainsi un moyen fiable d'assurer une traduction, de déclarer des objets Javascript ou encore d'attacher des méthodes à ces derniers, le tout en restant dans le monde OCaml. La translation assurée, nous pouvons nous concentrer sur la modélisation d'Enyo.

Le point critique de l'interfaçage se situe dans l'utilisation des traits dynamiques de Javascript, par exemple, utiliser la méthode *setContent(value)* sur un *kind : "Control"* qui ne possèdera cette méthode qu'après instantiation est rédibitoire pour une transcription directe en OCaml.

Après considérations, j'ai choisi de m'arrêter sur un modèle reposant sur une différenciation d'un objet en construction de celui d'un objet instancié. On représente ainsi un objet *Enyo* en OCaml grâce aux propriétés le définissant.



L'extension de l'objet en cours de création avec propriétés et méthodes personnalisées est abandonné. Cet aspect d'Enyo est nécessaire pour la partie logique de l'application, ce dont nous pouvons nous abstraire puisque nous souhaitons un modèle de calcul reposant sur un programme OCaml classique.

A partir de là, les objets de l'interface de programmation (API) d'Enyo et leurs méthodes sont suffisants si l'on considère utiliser Enyo comme une interface graphique multi-plateforme pour OCaml. Il est également nécessaire de pouvoir traiter les événements utilisateurs pour bénéficier d'une interface dynamique et réactive.

Dans un premier temps (3.1), nous présenterons la structure choisie, son fonctionnement et les raisons de cette décision.

Ensuite, la définition des propriétés et leur implémentation (3.2)

Enfin, nous nous attarderons sur les événements (3.3).

3.1 Représentation des objets

3.1.1 Modélisation

D’après le modèle choisi, on distingue donc la représentation en deux types. Le premier, à créer par l’utilisateur, que nous nommerons “kind” par la suite. Et le second, résultant d’une instanciation, possédant toutes les méthodes décrit dans l’API Enyo, que nous appellerons “obj”.

Dans un soucis d’aisance, j’ai fait le choix d’annoter ces “objets” à l’aide de types fantômes. Associés aux variants polymorphes, on profite ainsi d’un sous-typage. Cela a pour avantage de permettre un partage des noms de méthodes appartenant à plusieurs objets. Le désavantage majeur de cette solution reste cependant la confusion liée à la génération des messages d’erreurs lors du développement d’une application.

Nous obtenons ainsi les deux types “kind” et “obj” qui seront utilisés avec l’ensemble de tous les type objets Enyo en précisant l’utilisation covariante du paramètre.

```
type any_id =
  [ 'CONTROL | 'INPUT | 'BUTTON (* | ... *) ]
type +'a kind (* constraint 'a = [< any_id] *)
type +'a obj
```

Les signatures de méthodes partagées prennent alors la forme suivante :

```
sig :
  (*...*)

  val setContent :
    [< 'BUTTON | 'CONTROL (* ... *) ] obj
    -> string
    -> unit

  (*...*)
end
```

Pour construire l’objet Enyo, ce dernier doit pouvoir contenir :

- Les propriétés de bases incluent dans l’API Enyo
- Les fonctions de traitement des différents événements
- La liste des sous-composants

Le choix s’est donc porté sur une structure de ce type :

```
type handler = Handler of string * (any_id obj -> any_id obj
  -> any_event obj -> bool)
type js_value = Int of int | String of string
```

```

| (*...*)
| Array of js_value list
| Component of any_id obj
type +'a kind = {id:string;
                  components: any_id kind list;
                  handler_list:handler list;
                  prop_list : (string * js_value) list
                }

```

J’incorpore à la structure un champ *id* pour pouvoir permettre une introspection sur les futurs objets instanciés (voir 3.3).

Quant à la représentation Javascript, elle est simplement représentée par un objet Javascript du module “Js.Unsafe” de la bibliothèque *js_of_ocaml*.

```

type +'a obj = Js.Unsafe.any

```

Cela permet un travail direct sur les différents appels à transmettre au monde Javascript.

Munis des structures, on peut désormais s’intéresser à la construction proprement dite de cette représentation OCaml d’un objet Enyo.

3.1.2 Création

Pour cette phase de création, j’ai décidé de m’inspirer du module Tk d’OCaml qui utilise les options pour les fonctions prenant un nombre important d’arguments omissibles.

Plusieurs autres solutions auraient été possible, telle qu’une liste d’association mais dont je n’ai pas su trouver de façon élégante pour permettre à l’utilisateur de l’employer aisément.

Ainsi, nous obtenons un constructeur, ici de bouton pour l’exemple, de cette forme :

```

val bouton:
  ?components:any_id kind list
  -> ?content:string
  -> ?showing:bool
  -> ?src:string
  -> ?canGenerate:bool
  (* ... *)
  -> ?ontap:([ 'BUTTON] obj ->
              any_id obj ->
              [ 'GESTURE] obj -> bool)
  -> unit -> [>'BUTTON] kind

```

donnant à l’utilisation, pour un bouton contenant un simple label, ceci :

```

let mon_bouton = bouton ~content:"Valider" ()

```

Quelques bémols, cependant, à cette manière de procéder :

- Le nombre d’arguments de chaque constructeur augmente au fur et à mesure que l’objet soit profondément situé la hiérarchie,
- Les messages d’erreurs générés deviennent rapidement indigestes,

- Une grande redondance parmi les différents constructeurs.

Finalement, le code du constructeur revient à inclure chaque argument optionnel passé dans différentes listes (typiquement, la liste de propriétés et celle de fonctions de traitement) en testant au préalable leur existence. En ajoutant la liste de composants quelques constantes, un “ID” pour l’introspection et le champ “kind” nécessaire à Enyo, l’objet est prêt à être instancié en tant qu’application ou à être passer en tant que sous-élément dans un autre objet par sa liste de “components”.

```
let button
  ?(components=[])
  ?content
  ?ontap
  (*...*)
  () =
let prop_list= ref [("kind", String "Button")]
and handler_list= ref [] in
(match content with
  Some v -> prop_list := ("content",String v)
                        ::!prop_list
  | None -> ());
(match showing with
  Some v -> prop_list := ("showing",Bool v)
                        ::!prop_list
  | None -> ());
(*...*)
(match ontap with
  Some v -> handler_list := Handler ("ontap",v)
                        ::!handler_list
  | None -> ());
{id="BUTTON";
 components=components;
 prop_list=(!prop_list);
 handler_list=(!handler_list)}
```

3.1.3 Instanciation

Après avoir obtenu notre représentation OCaml, il est nécessaire de le traduire en Javascript. C’est le rôle de la fonction *Instanciate*.

Cette dernière parcourt les différentes listes (propriétés, événements, et composants) et construit un objet Javascript formé selon les spécifications d’Enyo que nous passons à *enyo.kind* et retournant l’instance de celui-ci avec un nom par défaut si non-fourni.

Une application d’*instanciate* sur le “bouton kind” définit plus haut (3.1.2) équivaut en Javascript alors à :

```
1  enyo.kind({kind:"Button", name:"Default_name", content:"
    Valider", _onyo_id:"BUTTON"});
2  var obj_enyo = new Default_name();
3  return obj_enyo;
```

La fonction étant relativement indigeste, je détaillerai ici les manipulations exécutées :

1. La liste de propriétés est évaluée et chaque type de valeur est traduite si besoin en Javascript. Chaque propriété est ensuite injectée dans un nouvel objet Javascript à l'aide de *Js.Unsafe.set*
2. Les différentes fonctions de traitement présentes sont “enveloppées” dans l'objet en tant que méthodes dans l'objet avec un appel à *Js.wrap_meth_callback* prenant une fonction dont le premier argument est l'objet lui-même représentant le *this*. Il est aussi nécessaire de déclarer l'objet *handlers* pour assurer le traitement des événements. (Plus de détails : 3.3).
3. Récursion sur tous les sous-composants présents et transformation de la liste de ces derniers en un tableau Javascript à passer au champ “components” de l'objet.

Voici, une forme épurée de cette fonction :

```
let instanciate (kind:([< any_id] as 'a) kind) : 'a obj =
  let rec build_component_tree : 'a. ([< any_id] as 'a) kind
    -> Js.Unsafe.any = fun kind ->
    let js_obj = Js.Unsafe.new_obj (variable "Object")
    [||]
  in
  (* 1 *)
  ;
  (if kind.handler_list != [] then
    (* 2 *)
  );
  (if kind.components != [] then
    (* 3 *)
    let array_component = Array.of_list (List.map
      build_component_tree kind.components) in
    Js.Unsafe.set js_obj "components" (array
      array_component));
    Js.Unsafe.set js_obj "_onyo_id" (string (kind.id));
    js_obj in
    kind_it (build_component_tree kind)
```

Enfin, on passe cet objet à la fonction *enyo.kind* pour récupérer un objet Enyo dont les méthodes sont désormais appelables tel que *setContent* si l'objet hérite de *Control* dans la hiérarchie Enyo.

L'utilisateur est désormais en mesure d'afficher son application en effectuant un appel à la méthode *renderInto* (si l'objet en possède la méthode) sur l'objet instancié.

3.2 Gestion des propriétés et des méthodes

Pour pouvoir représenter les valeurs Javascript faiblement typées, il a été nécessaire de définir un ensemble de ces type valeurs. Un type somme est

suffisant :

```
type js_value = Int of int | String of string
               | Char of char | Float of float
               | Dom_node of dom_node | Bool of bool
               | Array of js_value list | Component of any_id
               obj
```

Note : j’ai trouvé préférable de représenter le tableau Javascript sous forme de liste, cette dernière étant plus usitée par le programmeur OCaml.

Les propriétés possédées par les objets Enyo sont implémentées en tant que “Published” (voir 2.5). Ces propriétés possèdent donc des accesseurs (get) et des modificateurs (set) ainsi que la fonction *<valuenam>Changed*. Il faut alors pour chaque propriété générer ces méthodes.

J’ai fait le choix d’omettre la possibilité de définir un *<valuenam>Changed* qui peut cependant, si la logique de l’application le nécessite, être remplacé par un simple appel de fonction au moment de la modification de propriété.

Au niveau de l’implémentation, j’ai ajouté ces accesseurs/modificateurs pour chaque propriété définie et appellable par tous les composants en héritant.

Afin d’assurer une bonne traduction Javascript/OCaml, il a fallu selon les types employés traduire les variables. Pour cela, l’API de “js_of_ocaml” propose les fonctions nécessaires :

```
val bool : bool -> bool t
val to_bool : bool t -> bool
(*...*)
val array : 'a array -> 'a js_array t
val to_array : 'a js_array t -> 'a array
```

Selon le type de la propriété, on s’assure alors d’effectuer correctement cette traduction :

```
let getContent this () =
  let value = Js.Unsafe.meth_call
              this
              "getContent"
              [] in (* Appel de la methode sur l’objet
                    Javascript *)
  to_string value (* conversion de la valeur js obtenu en
                  valeur caml *)
```

```

let setContent this chaîne1 =
  let _ = Js.Unsafe.meth_call
    this
    "setContent"
    [| Js.Unsafe.inject (string chaîne1) |] in (*
      conversion caml -> javascript *)
  ()

```

Les propriétés Enyo étant pour la plupart initialisées par défaut, je n’ai pas jugé urgent de gérer les cas de valeur “null”. Il faudrait, pour certains cas, fournir cette sécurité.

Une autre difficulté s’est posée concernant les méthodes polymorphiques. Une méthode de type *setProperty(name, value)* est difficile à représenter en OCaml, c’est pourquoi j’ai préféré laisser ce problème de côté afin de ne pas compliquer le typage.

Quelques propriétés ont ainsi été laissées de côté, comme par exemple celles prenant des objets Javascript non-Enyo en valeur.

```

1 //bounds : {left: _offsetLeft_, top: _offsetTop_, width:
  _offsetWidth_, height: _offsetHeight_}
2 this.setBounds({width: 100, height: 100}, "px");
3 this.setBounds({left: "100", top: "40", width: "10em", right: "30
  pt"});

```

Il conviendrait d’étudier chaque cas particulier et de fournir une équivalence sûre quitte à réduire les possibilités d’usage. Par exemple, fournir une fonction à quatre arguments qui n’agit sur une seule unité de mesure.

Enfin, certaines propriétés et arguments de méthodes peuvent changer de type selon leur usage et il n’est pas toujours agréable d’avoir à effectuer une conversion OCaml (ex : *string_of_int*) sur ces appels. Une solution pourrait être de dupliquer ces méthodes, par exemple, en les préfixant par le type. Ainsi, la propriété “value” de l’objet Enyo *Slideable* prenant un int, qui par ailleurs effectue une collision de nom avec celle de l’objet *Input* (voir 4), pourrait devenir *int_getInput* s’appliquant à l’objet *Slideable* et à ses descendants, tandis que *Input* posséderait une méthode *string_getInput*.

3.3 Gestion des événements

3.3.1 Fonctionnement

Dans Enyo, la spécification des événements et de leur fonctions de traitement gardent toujours la même forme.

```

1 fonctionTraitement: function(emetteur, evenement){
2   //traitement
3   return bool; }

```

La fonction attachée prend en paramètres l'émetteur de cet événement, c'est-à-dire, le composant qui a généré ou propagé celui-ci, et l'objet événement lui-même. Celui-ci retourne un booléen spécifiant si la propagation doit être effectué. (voir : 2.4 pour plus de détails)

Pour adapter l'événement en OCaml, j'ai choisi ce type :

```
type any_event = [ 'GESTURE (* | Ensemble des evenements *) ]
type handler = Handler of string * (any_id obj -> any_id obj
-> any_event obj -> bool)
```

La gestion de l'événement est donc représentée par un couple contenant le nom de l'événement et la fonction de traitement associée avec en premier argument le *this* de la méthode (voir : 2).

En informant la signature dans chaque constructeur, il devient aisé de constituer un typage correct. Ainsi, nous pouvons résoudre le *this* selon l'objet en cours de construction et l'événement attendu. L'émetteur ne peut cependant pas être résolu, nous y reviendrons plus loin.

Pour un bouton, nous attendrons un argument de ce type :

```
val bouton:
  (* ... *)
  -> ?ontap:([ 'BUTTON ] obj -> any_id obj -> [ 'GESTURE ] obj ->
    bool)
  -> unit -> [ >'BUTTON ] kind
```

A l'utilisation, nous pourrons alors utiliser les méthodes de bouton sans causer de conflit de type puisqu'employé dans un contexte correct :

```
let traitement_bouton this emetteur evenement =
  setContent this "Nouveau_contenu";
  setDisabled this true;
  true

let mon_bouton = bouton ~content:"contenu" ~ontap:
  traitement_bouton ()
```

J'ai défini les événements de la même manière que les objets. Il n'est, en revanche, pas nécessaire de partager les méthodes applicables sur les événements puisqu'ils ne sont pas liés entre eux. Il est cependant nécessaire d'affecter à chaque type d'événements des fonctions d'accesseurs pour récupérer les informations contenus.

Par exemple, pour l'événement souris, rebaptisé "gesture" pour se conformer à Enyo, les accesseurs, stipulés dans la définition IDL W3C, sont définis :

```
val gesture_screenX : [ 'GESTURE ] obj-> int
val gesture_screenY : [ 'GESTURE ] obj-> int
val gesture_identifieur : [ 'GESTURE ] obj-> int
val gesture_detail : [ 'GESTURE ] obj-> int
(* ... *)
```

À l'instanciation d'un objet pourvu d'un événement, il est nécessaire de passer l'événement traité dans un objet contenu dans le champ *handlers* de

celui-ci. Ainsi, le *mon_bouton* définit plus haut, aura cette forme en Javascript :

```
1 {kind:"Button", handlers:{ontap:"tap"},
2   tap:function(){ return callback_caml(this) /* gere par
   js_of_ocaml */ }}
```

Cette stratégie permet à chaque composant de posséder sa fonction de traitement mais présente un gros soucis lors du “bubbling” (remontée d’événement). En effet, il faut pouvoir connaître l’origine de l’événement à tout moment .

Prenons une situation d’exemple en Enyo avec notre représentation :

```
1 enyo.kind({
2   kind:"Control",
3   name:"App",
4   components:[
5     {kind:"Control",
6       components:[
7         {kind:"Button", content:"Vert",
8           handlers:{ontap:"tap"},
9           tap:function(emetteur, event){
10             //emetteur = this
11             return false; //propagation au noeud-parent
12           }
13         },
14         {kind:"Button", content:"Bleu",
15           handlers:{ontap:"tap"},
16           tap:function(emetteur, event){
17             //emetteur = this
18             return false; //propagation au noeud-parent
19           }
20         }
21       ]
22     },
23   ],
24   handlers:{ontap:"tap"},
25   tap:function(emetteur, event){
26     //emetteur = sous-control
27     // change le fond en noir si l’emetteur est b1
28     // ou bien en blanc si b2
29     alert(emetteur);
30     return true;
31   }
32 })
```

Ici, nous voulons, si l’on clique sur l’un des deux boutons, changer la couleur du fond du “control” racine selon le bouton cliqué.

Il faut donc pouvoir tester au niveau racine, quelle a été l’origine des événements. Le problème ici provient de la propagation au composant parent par les boutons. En effet, l’argument *emetteur* reçu par la racine ne contient pas l’un des deux boutons mais le “Control” intermédiaire par lequel l’événement a transité.

En Enyo, un simple : *emetteur.originator* renverrait le bouton ayant généré l'événement, et nous permettrait de déterminer la source mais l'implémentation de cette propriété aurait peu de sens dans la représentation actuelle puisqu'il impliquerait la présence d'une propriété *originator* dans tous les objets manipulés, même en dehors d'un événement.

Une solution à ce problème peut être de conserver, dans son application, une valeur référencée contenant l'objet ou l'information nécessaire pour effectuer le traitement souhaité. Je propose une autre stratégie plus loin : 3.3.3.

3.3.2 Introspection

L'implémentation actuelle de cette stratégie de traitement pose un problème pour typer l'émetteur. Pour pallier à cela, j'ai fourni lors de la construction des objets un champ supplémentaire¹ à chaque "kind" qui va s'étendre à l'objet Javascript : *_onyo_id*.

Munis de cette propriété, nous pouvons désormais accéder à ce champ pour le typer correctement. C'est le rôle de la fonction *as_a* :

```
val as_a : (< any_id] as 'a) -> [< any_id] obj -> 'a obj
```

Si l'objet n'est pas de l'"id" spécifié, une exception est levée.

Avec ceci, nous pouvons agir sur l'émetteur :

```
let mon_bouton = ~content:"Bouton" ~tap:(fun _ _ _ -> false)
()

let traitement this emetteur evenement =
  try
    let bouton_emetteur = as_a 'BUTTON emetteur in
      (* bouton_emetteur : ['BUTTON] obj *)
      setDisabled bouton_emetteur true;
      true
  with Bad_kind -> true

let mon_control = ~components:[mon_bouton] ~ontap:traitement
()
```

Dans cet exemple, le bouton capture l'événement *ontap* mais le délègue à son composant racine. Ce dernier essaye de transtyper l'émetteur en bouton et le désactive.

Cette solution reste cependant inélégante et assez contraignante pour le développeur.

3.3.3 Autre stratégie de propagation

Dans un développement Enyo, il est courant de ne définir les fonctions de traitement qu'à la racine de l'arbre de composants. Ceci permet d'avoir

1. Le champ *kind* étant effacé à l'instanciation Enyo, il n'est pas possible d'en faire usage

un accès à tous les sous-composants grâce à la table de hash que possède le composant principal. Nous obtenons ainsi le programme reprenant l'exemple posant problème :

```

1  enyo.kind({
2    kind:"Control",
3    name:"App",
4    components:[
5      {
6        kind:"Control",
7        components:[
8          {kind:"Button", name:"b1", content:"Vert",
9            ontap:"tap"
10         },
11         {kind:"Button", name:"b2", content:"Bleu",
12           ontap:"tap"
13        }
14      ]
15    }
16  ],
17  tap:function(emetteur, event){
18    if (this.$.b1 == emetteur){
19      // couleur noire
20    } else if (this.$.b2 == emetteur){
21      // couleur blanche
22    }
23    return true;
24  }
25  //Il peut aussi etre interessant de definir une fonction
26  //pour chaque bouton :
27  tap_b1:function(em, ev){
28    //couleur noire
29    return true;
30  }
31  tap_b2:function(em, ev){
32    //couleur blanche
33    return true;
34  }
35 })

```

Un comportement semblable pourrait être implémenté résolvant par la même occasion, le typage de l'émetteur.

Plutôt que de définir les fonctions de traitement au niveau des objets, on pourrait les définir sous des labels uniques, à la racine. Nous obtiendrions, une signature dans le constructeur pour le traitement de ce type ressemblant à :

```

button ?content:string
(* ... *)
?ontap:([ 'APPLICATION' | obj ->
          [ 'BUTTON' | obj ->
            [ 'GESTURE' | obj -> bool )

```


Il conviendrait à priori de définir un nouvel objet “Application” sur lequel il serait possible de fournir une lecture de la table de hash et ainsi d’agir sur les différents objets.

L’instanciation à même titre retournerait un : *[‘APPLICATION’ obj.*

J’essayerai de convenir de la validité d’une telle approche et de son implémentation dans une version future.

Chapitre 4

Langage de Description d'Interface (IDL)

Enyo fournit une API suffisamment grande (une centaine d'objet environ) pour qu'il devienne rébarbatif d'implémenter ceci à la main. L'implémentation d'un IDL s'est donc révélé être une bonne option pour pouvoir représenter facilement la hiérarchie de composants.

OCaml étant un outil tout indiqué pour ce type de travail, je n'ai pas eu à faire de parallèle avec d'autres langages (mis à part quelques scripts Perl pour parser la hiérarchie).

En premier lieu, il convient de présenter les structures choisies pour abriter les objets. Puis, j'expliquerai brièvement la génération du code et les problèmes rencontrés.

4.1 Interface des objets

Un objet Enyo présente 3 grandes parties distinctes :

- Les méthodes,
- les propriétés,
- les événements qu'il est capable de "capturer".

Pour les méthodes et les propriétés, il est nécessaire de fournir le type à l'IDL pour assurer à OCaml la sûreté du typage.

Les événements, quant à eux, possèdent un lien vers le type d'événement et les champs qu'il faut préciser.

```
type values_rep = String | Int | Float | Bool | Unit
                | Component | Array of values_rep | Dom_node

type method_rep = Method of string * values_rep list

type attributes_rep = Attribute of string * values_rep * bool (*
    bool => Generate propChanged function? *)
```

```

type event_rep = Event of string * attributes_rep list

type handler_rep = Handler of string * event_rep

type object_rep = Type of string * method_rep list *
  attributes_rep list * handler_rep list
  | Type_gen of object_rep * string * method_rep
    list * attributes_rep list * handler_rep
    list
  (* Le deuxieme element de la somme n'a plus de
     reel utilite, mais n'ayant pas eu le temps
     de retravailler le code, il est a ignorer *)

```

J'ai donc choisi de représenter l'objet Enyo par :

- Son nom Enyo,
- une liste de représentation de méthodes,
- une liste de représentation de propriétés¹,
- une liste des noms d'événements que ce type d'objet est capable de traiter.

On obtient ainsi pour l'objet "Control" d'Enyo, la représentation suivante :

```

Type ( "Control",
  [
    Method ( "render", [Unit; Unit]);
    Method ( "rendered", [Unit; Unit]);
    Method ( "renderInto", [Dom_node; Unit]);
    Method ( "setBounds", [Int; Int; Int; Int; Unit]);
    Method ( "show", [Bool; Unit]);
    Method ( "write", [Unit; Unit]);
    (* ... *)
  ],
  [
    Attribute ( "style", String, true);
    Attribute ( "content", String, true);
    Attribute ( "allowHtml", Bool, false);
    Attribute ( "src", String, true);
    (* ... *)
  ],
  [Handler ( "ontap", Gesture_event._gesture)]
)

```

En étudiant l'API, j'ai pu transcrire les méthodes et propriétés en m'efforçant de typer au plus juste. En revanche, quelques libertés ont été prises :

- Je n'ai pas jugé utile l'ajout de méthodes trop polymorphes telle que : *setProperty(name, value)*.
- Par manque de documentation de l'API sur certaines, il a été quelques fois nécessaire de compléter moi-même les signatures de méthodes en

1. L'utilisation du booléen pour savoir si une fonction de type <value>Changed doit être implémenté est temporairement abandonné, pour les raisons spécifiées dans la partie de gestion des propriétés 3.2

effectuant des tests en restant toutefois rigoureux.

- J’ai omis d’inclure les méthodes protégées, pour la simple raison que nous ne traitons pas l’extension objet, ici.

Je complète ainsi chaque objet et l’inclus dans un arbre d’`object_rep` recréant ainsi la hiérarchie Enyo nécessaire permettant à chaque composant d’hériter des méthodes et attributs. La hiérarchie désormais complète, la génération de code peut débuter.

4.2 Générateur de code

L’objectif de cette génération de code est d’obtenir en sortie un module complet basé sur la représentation IDL avec le modèle choisi 3.

La difficulté de cette étape a été le parcours de l’arbre pour résoudre les dépendances :

- Un objet doit transmettre à ses fils toutes les propriétés et “handlers” d’événements qu’il possède afin que leurs constructeurs soient en mesure de les proposer à l’utilisateur
- En revanche, les méthodes auront besoin de la liste de tous les identifiants de sa descendance pour pouvoir spécifier les méthodes et permettre aux enfants de les appeler

Une simple récursion sur l’arbre permet l’obtention d’un résultat satisfaisant.

Plus tard, j’ai constaté qu’un cas spécial ne permettait pas un parcours aussi banal : lorsque deux objets définissent des méthodes ou propriétés de même noms sur des branches différentes de l’arbre, il y a collision de noms et il est impossible de factoriser simplement les dépendances².

Afin de résoudre ceci, je réalise des parcours préalables en remplissant des tables de hachage contenant en clé le nom de méthodes et de propriétés avec en valeur leurs listes des dépendances. J’applique ensuite la récursion, en consultant la table de hachage pour résoudre les incohérences.

Note : On aurait également pu “tagger” les différentes incohérences au moment de la représentation de l’objet en spécifiant que cette méthode ou propriété est partagée avec un autre objet. Plusieurs solutions sont possibles...

L’arbre de résolutions de dépendances construit, il ne reste plus qu’à réaliser les impressions pour pouvoir générer le module OCaml prêt à l’emploi.

2. J’ometts le fait que deux méthodes de même nom peuvent être de types différents, ce qu’il faudrait traiter si le cas était présent

Chapitre 5

Conclusion

A travers ce projet, nous avons pu établir un certain nombre de choses. Tout d'abord, nous avons pu établir que le style de programmation d'Enyo nous offrait un nombre important de possibilités. Il a fallu étudier ce style, son fonctionnement et en abstraire les concepts afin de définir un modèle intéressant et cohérent.

La scission de la représentation et de l'instanciation de l'objet Enyo sur lequel j'ai basé mon modèle répondent à l'objectif de sûreté que nous nous étions fixé tout en assurant un degré important de liberté vis-à-vis de l'API d'Enyo.

Enfin, la définition d'un IDL a permis une génération fiable de l'implémentation finale mais aussi de permettre de tenir à jour l'API Enyo de manière assez aisée en cas de besoin.

Cependant, le rendu final subit quelques défauts :

- L'utilisation de variants, bien que permettant une bonne modélisation, peut être perçue comme trop complexe pour une prise en main rapide. Cela fournit également des messages d'erreurs trop agressifs. Cette dernière remarque s'applique aussi pour l'utilisation des paramètres optionnels lors de la construction des objets.
- L'accession aux différents composants instanciés lors de la définition du traitement des événements n'est pas satisfaisante et gagnerait à être améliorée.
- L'IDL ne traite pas certains cas particuliers et reste relativement primaire.

Pour permettre une continuation du projet, il peut-être intéressant de reprendre les problèmes énoncés avec, par exemple, les différentes approches énoncées dans les différentes sections. Pour résumer :

- Réfléchir à une représentation différentes que les variants, quitte à rendre le module plus lourd, par une approche objet ou type simple.
- Pouvoir accéder aux instanciations Enyo par le "kind" de l'objet, en s'assurant de son existence. Par exemple, ainsi :

```

let bouton_obj = getObj bouton_kind (* 'a kind -> 'a
    obj *)
setContent bouton_obj "Nouvelle_valeur"

```

- Il serait bien de compléter la hiérarchie en fournissant des solutions adéquates aux cas particuliers. Ensuite, pouvoir fournir des possibilités d’extension à l’IDL en permettant des définitions internes de composants. Par exemple, être capable de créer un nouvel objet “ChampValidation” possédant deux composants “Input” et “Button” ainsi qu’une fonction de validation.
- Il faudrait réfléchir à une représentation des noeuds du DOM. Est-ce que l’on souhaite passer par “js_of_ocaml”, fournir une chaîne le représentant, ou encore définir quelques noeuds principaux et brider leur utilisation ?

Sur le plan personnel, ce stage m’a été, je pense, très profitable ; j’ai pu consolider mes bases en OCaml ainsi que mon engouement pour ce langage. Je le dois en bonne partie à l’équipe capable et dynamique dont j’ai eu la chance d’être entouré pendant ces deux mois. De plus, cette expérience au sein d’un laboratoire m’a sérieusement fait réfléchir à une éventuelle formation vers la recherche. Je me satisfais donc d’avoir pu effectuer ce stage dont j’en sors enrichi.

Chapitre 6

Annexes

6.1 Hiérarchie complète Enyo

```
|-- Layout
|   |-- Arranger
|   |   |-- CardArranger
|   |   |   |-- CardSlideInArranger
|   |   |-- CarouselArranger
|   |   |-- CollapsingArranger
|   |   |-- LeftRightArranger
|   |   |-- TopBottomArranger
|   |-- BaseLayout
|   |-- FittableLayout
|-- Object
    |-- Async
    |   |-- Ajax
    |   |-- JsonRequest
    |-- Component
        |-- _AjaxComponent
        |   |-- WebService
        |-- Animator
        |-- DragAvatar
        |-- ScrollMath
        |-- Selection
        |-- Signals
        |-- UiComponent
            |-- canvas.Control
            |   |-- canvas.Image
            |   |-- canvas.Shape
            |       |-- canvas.Circle
            |       |-- canvas.Rectangle
```

```

|          |-- canvas.Text
|-- Control
|   |-- Canvas
|   |-- FittableColumns
|   |-- FittableRows
|   |-- FlyweightRepeater
|   |-- Group
|       |-- onyx.RadioGroup
|-- GroupItem
|   |-- ToolDecorator
|       |-- Button
|           |-- onyx.Button
|           |-- onyx.MenuItem
|           |-- onyx.InputDecorator
|-- Image
|-- Input
|   |-- Checkbox
|       |-- onyx.Checkbox
|   |-- RichText
|       |-- onyx.RichText
|   |-- TextArea
|       |-- onyx.TextArea
|-- Node
|-- onyx.Drawer
|-- onyx.Grabber
|-- onyx.Groupbox
|-- onyx.GroupboxHeader
|-- onyx.Icon
|   |-- onyx.IconButton
|-- onyx.Item
|-- onyx.MoreToolbar
|-- onyx.ProgressBar
|   |-- onyx.ProgressButton
|   |-- onyx.Slider
|-- onyx.Scrim
|-- onyx.Spinner
|-- onyx.ToggleButton
|-- onyx.Toolbar
|-- onyx.TooltipDecorator
|   |-- onyx.MenuDecorator
|       |-- onyx.PickerDecorator
|-- Option
|-- OwnerProxy
|-- Panels

```



```

|-- Popup
|   |-- onyx.Popup
|       |-- onyx.Menu
|           |-- onyx.Picker
|               |-- onyx.FlyweightPicker
|                   |-- onyx.Tooltip
|-- Repeater
|-- Scroller
|   |-- List
|       |-- PullDownList
|-- ScrollStrategy
|   |-- TouchScrollStrategy
|       |-- TranslateScrollStrategy
|-- ScrollThumb
|-- Select
|-- Slideable

```

6.2 Déploiement d'applications

Le déploiement d'applications sur les différentes plateformes nécessite quelques changements mineurs de l'application en fonction de la cible visée.

Tout d'abord, Enyo fournit un “bootplate” décrivant l'organisation standard d'une application. Ce dossier contient un script “deploy” permettant de préparer l'application au déploiement ainsi que l'analyse du programme et la suppression des fichiers inutilisés.

A partir de là, pour un déploiement sur tablette HP, par exemple, il est nécessaire d'obtenir les outils développeur de la tablette permettant de “packager” l'application. Pour un déploiement sur téléphone (“android”, “iOS”, etc.), il est nécessaire de télécharger “PhoneGap” [lien] qui propose ce genre de “packaging”. Il est toutefois nécessaire de fournir un noeud *meta* dans le document Html principal de l'application selon la plateforme. Les détails sont accessibles à cette adresse :

<https://github.com/enyojs/enyo/wiki/Platform-Specific-Deployment>

6.3 Exemples de programme

Les exemples d'applications se situent dans le dossier “exemples” à la racine du projet.

On pourra y trouver quelques applications JavaScript ainsi que des jeux utilisant la bibliothèque inspiré des exemples contenu sur le cédérom du manuel “DAOC”[2].

Bibliographie

- [1] Benjamin Canou. *Programmation Web Typée*. PhD thesis, Université Pierre et Marie Curie, 2011. [lien].
- [2] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'Applications avec Objective Caml*. O'Reilly France, 2000. [lien].
- [3] Développeurs Enyo. *Documentation Enyo 1.0*. HP, 2011. [lien].
- [4] Développeurs Enyo. *Documentation Enyo 2.0*. HP, 2012. [lien].
- [5] Ocsigen Team. *Js_of_ocaml*. Ocsigen, 2010-2012. [lien].