SARL MEZZÒNOMY

Programmation informatique (6201Z)
34 Place de Catalogne 31700 BLAGNAC
Toulouse B 504 641 473

Méthodologie BHB

Table des matières

Introduction – Contexte

La **blockchain hypertexte** est une technique originale et unique de construction de systèmes coopératifs décentralisés et sécurisés. Comme son nom l'indique, notre technologie emprunte à deux champs majeurs de l'informatique contemporaine :

- **blockchain** : un ensemble de méthodes décentralisées de cryptographie répartie permettant de tenir un registre de « *briques* » partagées, inviolables et éternelles sans organe central de contrôle[*BLOCKCHAIN*].
- **hypertexte** : un ensemble de méthodes décentralisées de mises en relation de la connaissance retardant la dégradation de l'information dans le temps[*WWW*].

La fertilisation croisée de ces deux techniques relève presque du bon sens :

sachant qu'une brique est pensée comme « *éternelle* », mieux vaut l'écrire avec des méthodes conçues pour résister au temps !

La **méthodologie BHB**, dont ce papier présente la première version, permet à des ingénieurs métiers *sans aucune notion en cryptographie répartie* de produire des applications générant des blockchain hypertextes avec toutes les bonnes propriétés.

Pour ces ingénieurs, disposés à apprendre un peu d'hypertexte, la **blockchain hypertexte** se résumera alors à une réinterprétation de la notion du temps à la fois :

- dans le « *processus implémenté* », appelé le **test**
- et dans le « *processus de son implémentation* », appelé le **code**.

Introduction – Règles temporelles

En ce qui concerne le **test**, le temps y est représenté de manière extrêmement concrète en blockchain hypertexte :

- Tout **test** est une séquence ordonnée (= un « *fichier* ») de briques.
- La seule opération possible sur un **test** est la « *continuation* », soit ajouter une ou plusieurs briques à la fin du fichier.

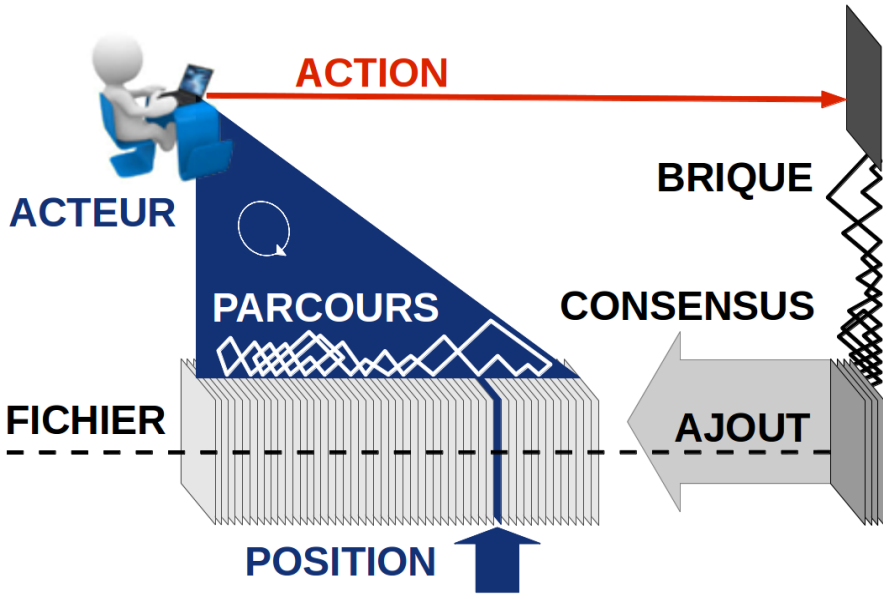


Figure 1. La position de l'acteur

Cette configuration de **test** correspond à une simplification de la situation de l'acteur. Pour le **testeur**, la partie droite est un « *rebouclage* » : **AJOUT = ACTION**. Cette simplification permet à l'ingénieur de se concentrer sur l'**élaboration** des briques, de se dégager des contraintes liées au **consensus** et au **scellement**.

Néanmoins, ces contraintes ne disparaissent pas complètement mais peuvent se comprendre comme des règles qualité, instruisant un rapport au temps compatible avec l'objet. Nous avons tenté de synthétiser nos premières expériences dans l'accompagnement vers la blockchain hypertexte grâce à trois règles temporelles :

Règle « de plomb » :

Toute brique est atomique, inviolable et éternelle.

*< !-- **Blockchain** : les briques sont stockés dans des blocs, scellés sur la chaîne -->*

*< !-- **Hypertexte** : une briques est un **XML** et ses liens avec l'existant sont des **URL**, le langage de consensus est **XSL**.-->*

Règle « d'or » :

Faites le simple, le complexe est ailleurs.

<!-- Le complexe : une brique est un fragment d'état, comme le morceau d'un puzzle. Surtout pas un message, ni une instruction, ni un ordre. Par exemple, en XSL, il n'y a pas d'affectation.-->

Règle « du soupir » :

L'histoire est longue, surtout au début.

<!-- Plus on s'intéresse au fondamentaux de cette histoire, plus on progresse vers son commencement, plus cela devient technique. Le complexe est au commencement. La page blanche est un espace hostile. -->

<!-- A chacun son temps. Ne pas chercher à partir de la page blanche. Pour ne pas se perdre, remonter le temps jusqu'à rencontrer les interlocuteurs adéquats pour le bon niveau de complexité. -->

Cette dernière règle contient le fondement philosophique de la méthode.

Introduction – Phases d'un test

> ***Le** code **a pour objet de créer une « histoire » dont le** test **est la continuation.***

Nous identifions quatre phases dans lesquelles un **test** peut se trouver, notre attention sera principalement focalisée sur les deux phases médianes :

- la phase **graphique**, où le scénario est décrit sous une forme de story-board.
- la phase **statique**, où les scénario est décrit sous forme de briques de données et d'une feuille de style (XSL) élémentaire.
- la phase **dynamique**, où l'XSL décrit aussi l'ensemble des briques possibles à partir de l'état courant.

  <!-- A ce stade, et à la fin de ce stade seulement, l'histoire s'écrit de manière autonome depuis le « *début du test* ». -->

- la phase **physique**, où l'XSL est lui même assemblé à partir de briques existantes, là l'histoire en entier est écrite de la manière la plus compacte possible au sein de toutes les histoires en cours.

  <!-- Le **test** est archivé et sa maintenance déléguée aux développeurs. Les développeur ont pour mission de réécrire l'histoire en garantissant que tous les tests soient préservés. -->

Différentes parties d'un **test** peuvent être dans des états de maturité différent, Rien n'empêche un **test** où un fragment de régresser.

<!-- Cela arrive quand les phases aval amènent des contraintes temporelles jusque là inconnues, et ces contraintes peuvent imposer de reprendre les phases amont. -->

Cette cartographie de la maturité d'un **test** est un outil essentiel à la coopération des ingénieurs.

Introduction – Story-board

Considérons la séquence suivante comme un exemple de **test** sous sa forme première, le story-board :

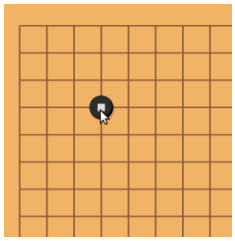
*<!-- Notre exemple sera une **partie de jeu de go**, qui pose, dans des termes (métiers) dont le sens peut être trouvé simplement sur internet toutes sortes de problèmes propres au systèmes transactionnels. -->*

<?act tom initie une **partie de jeu de go** en 19x19.?>

<?act tom rejoint la **partie** comme **noir**.?>

<?act thierry rejoint la **partie** comme blanc.?>

<?act **noir** pose une pierre en hoshi.?>



*<**bhb**:expect href="example-go-001.png"/>*

<!-- Ici, soupir, le « block » englobant contenant les informations de provenance a été omis pour éviter d'encombrer la présentation avec des notions inutiles à cet endroit. -->

***Tout** test **est exprimé en langage méta-étendu (ang. XML pour « eXtended Meta-Language »)**.*

Nous rappelons qu'un fichier **XML** a pour vocation d'être « *imprimable* » et cette contrainte irrigue ce document et les conventions qu'il véhicule.

- Les **comments** sont libres.

  <!-- Notre exemple sera une **partie de jeu de go**, qui pose, dans des termes (métiers) dont le sens peut être trouvé simplement sur internet toutes sortes de problèmes propres au systèmes transactionnels. -->

- les **processing-instructions** servent a décrire des actions sous une forme informelle.

  <?act **noir** pose une pierre en hoshi.?>

  <!-- Nous pourrons utiliser d'autres catégories (model, view, ...) dans le processus de réalisation. -->

- Le tag ***bhb**:expect* permet de définir un attendu dans l'attribut *href*.

```
<bhb:expect href="example-go-001.png"/>

<!-- Toutes sortes de fichiers sont acceptés, les fichier HTML permettent la la convergence assistée par différentiel et la validation automatique. -->

<!-- Les attendus doivent pouvoir être présentés sous forme de vignettes graphiques.  -->
```

- Les autres tags verront leur sens définit par une XSL qui produira l'HTML résultat présenté à l'utilisateur à chaque instant. Le **code** utilise les **namespaces** suivants :

```
xmlns:bhb="ptp://hypertext.blockchain"

xmlns:on="http://hypertext.blockchain/events"

xmlns:html="http://www.w3.org/1999/xhtml"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

- Chaque métier utilise ces propres **namespaces**, par exemple nous définissons le namespace du jeu de go par une URL désignant un site dont le comportement nous sert de référence.

```
xmlns:go="http://www.playok.com"
```

Introduction – Etat de l'art

Tout navigateur hypertexte standard (Chrome, IE, Safari, …) permet de réaliser une séparation modèle-vue en utilisant des **XSL**. Ce sont des techniques légères, largement répandues, immédiatement disponible, mais ne répond pas aux exigences d'un système blockchain. Bien que dans certaines situation, comme fournir un poste de travail **CSS** pour des *graphistes*, ce type de solution puisse avoir des applications ponctuelles, cet état de l'art est avant tout un requête formulée auprès de l'ingénieur :

> ***Le** code **ingénieur est exprimé en « langage de feuilles de styles étendues » (ang. XSL pour « eXtended Stylesheet Language »)**.*

Pour notre exemple, un XML aurait la forme suivante, nous l'appelons le « *modèle* » :

```
<?xml version='1.0' encoding='iso-8859-1'?>
<?xml-stylesheet type='text/xsl' href='XSL'?>
<go:game>
    <?model go-ban 19x19?>
    <go:black>
        <bhb:user login="tom" passwd="<?model SHA256 login + passwd?>"/>
        <go:stone>
            <?model hoshi?>
        </go:stone>
        <?model cet utilisateur a de multiples pierres?>
    </go:black>
    <?model un autre utilisateur, go:black, reproduit ce type de déclaration?>
</go:game>
```

XSL aurait la forme suivante, nous l'appelons l'« *application* » :

```
<?xml-stylesheet type='text/xsl' href='SDK'?>
<xsl:stylesheet>
    <xsl:template match="go:game">
        <?view dessiner la grille?>
        <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="go:white/go:stone">
        <?view dessiner une pierre blanche selon sa position?>
    </xsl:template>
    <xsl:template match="go:black/go:stone">
        <?view dessiner une pierre noire selon sa position?>
    </xsl:template>
</xsl:stylesheet>
```

La vue est produite par l'application du modèle à la position de l'utilisateur. Dans le cas du jeu de go, jeu à information complète, la position n'a pas d'impact sur la vue statique. La différence selon la position concernera la phase dynamique, où un seul des deux joueurs est alternativement en situation de poser une pierre.

A ce stade, nous voyons que la phase statique tournera autour d'une question : « *comment gérer le* ***go****:goban ?* ». Pour le moment, cette question se pose uniquement de manière *graphique*, mais il est probable que les autres phases s'y expriment puisque toute la règle du jeu de go finira par s'y exprimer sous cette forme.

Phase Statique – Réécritures

L'essentiel de la démarche consiste à considérer tout fichier hypertexte comme résultat d'une histoire, composée de briques éternelles et inviolables, ce que nous avons appelée la règle de plomb.

Corollaire de la règle de plomb : ***Pour qu'une brique soit modifiée ou enlevée, une autre brique est nécessaire.***

Ce type de contrainte, tous les juristes la connaissent, dans leur vocabulaire, cela devient « *pour qu'une loi soit amendée ou abrogée, une autre loi est nécessaire.* ». Les mêmes causes produisent les mêmes effets. Comme une loi est (presque) toujours la réécriture d'une autre loi, une brique sera (presque) toujours la réécriture d'une autre brique.

Le (presque) concerne, bien évidemment, les questions relatives au début de l'histoire et donc, le complexe.

Pour arriver à décrire le système sous forme d'une suite de briques, nous posons que **chaque brique est une réécriture d'un hypertexte**. Considérons la brique suivante, identifiée par la suite comme la brique 0 de notre séquence :

```
<!-- ici nous utilisons la règle du soupir, le début de l'histoire est le complexe,  et la règle d'or nous impose de n'y pas regarder en priorité -->

<go:game bhb:id="0">

    <?model go-ban 19x19>

    <go:black>

        <bhb:user login="tom" passwd="<?model SHA256 login + passwd?>"/>

    </go:black>

</go:game>
```

Ajouter une pierre en « *hoshi* » s'obtient en appliquant la brique suivante,

appelée 1 :

```
<bhb:link append="0/go:black" bhb:id="1">
    <go:stone>
        <?model hoshi?>
    <go:stone>
</bhb:link>
<!-- nous réduisons les urls de liens à des « paths », l'utilisation des autres composantes de la norme URL sont laissés à des documentations plus avancées -->
```

La partie soulignée est une URL qui identifie de manière unique un nœud DOM d'*une* brique antérieure, ici le nœud **go**:black la brique 0. Dans ce contexte, l'opération correspond à l'ajout d'un « *child* » à un nœud donné et l'insertion de la brique 1 à l'adresse *0/**go**:black* produit un hypertexte classique :

```
<go:game>
    <?model go-ban 19x19>
    <go:black>
        <bhb:user login="tom" passwd="<?model SHA256 login + passwd?>"/>
        <bhb:link append="0/go:white">
            <go:stone position="dd"/>
        </bhb:link>
    </go:black>
</go:game>
```

Par la « ~~*disparition*~~ » des **bhb**:link, nous signifions leur sémantique :

- lorsqu'une brique est active, les liens sont « *traversés* » et « *invisibles* ».
- Lorsqu'une brique est inactive, les liens sont « *évités* » (et toujours « *invisibles* »).

Ce qu'il essentiel de comprendre ici, c'est l'ampleur de la créativité qui peut s'exprimer dans la **décision** d'activer un **bhb**:link :

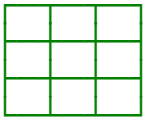
- La notion de **version** relève bien évidemment de cette notion, l'ensemble des bhb:link formant une structure d'événements [EVENTS] dans laquelle toute « configuration » est une version admissible (Cf. Annexe TED)
- La notion de **droit** relève aussi de cette notion, suivant les droits de celui qui interprète la blockchain certains liens peuvent être inhibés. Ceci n'a pas de sens dans un environnement avec une seule blockchain (Cf. Phase Dynamique – Consensus) mais dans un environnement multi-blockchain[BREVET], si (Cf. Phase Dynamique – Initiation).
- La notion de **latence** enfin, à la base des *smarts-contracts*, relève aussi de cette question. Une réécriture peut être valable pendant un certain temps, à partir d'une certaine date, etc. etc.

Phase Statique – Symboles

Dans l'exemple de « l'ajout » d'une pierre, la brique a un seul nœud ***bhb**:link* et son application sur un hypertexte « *plat* » demeure un hypertexte « *plat* ».

Mais dans le cas général, si la brique a plusieurs nœuds ***bhb**:link*, comment représenter le fait que l'on ajoute des nœuds à différents nœuds, potentiellement à l'autre *bout* de l'hypertexte ? Comment cet hypertexte va-t-il se « *replier* » sur lui-même ?

Par exemple, supposons que nous voulions coder la topologie du **go**:goban dans la structure de donnée, comment faire ?



<?model go-ban 3x3>

<!-- La matrice est toujours une forme difficile à concilier avec des formes hypertextes où il faut toujours privilégier de manière arbitraire une dimension : lignes d'abord ? Colonnes d'abord ? Plus le système dispose de dimensions plus cette incapacité à traiter des systèmes multidimensionnel grève les systèmes hypertextes. -->

Nous étendons les hypertextes en introduisant un attribut bhb:symbol avec une valeur donnée en exactement deux exemplaires.

La première ligne du goban 3x3 s'écrira alors :

```
<go:go-ban>
    <go:stone>
        <go:link bhb:symbol="haa"/>
        <!-- aucun risque de confondre go:link avec bhb:link !-->
        <go:link bhb:symbol="vaa"/>
    </go:stone>
    <go:stone>
        <go:link bhb:symbol="haa"/>
        <go:link bhb:symbol="vab"/>
        <go:link bhb:symbol="hab"/>
    </go:stone>
    <go:stone>
        <go:link bhb:symbol="hab"/>
        <go:link bhb:symbol="vac"/>
    </go:stone>
    <?model compléter les lignes manquantes?>
    <!-- dès lors qu'il y a une routine, « il y a une application pour ça »...-->
</go:go-ban>
```

Cette extension de l'hypertexte aux données multidimensionnelles par le truchement des symboles va nous offrir la capacité de réaliser tous les systèmes transactionnels avec la « *blockchain hypertexte* », à commencer par le go.

Phase Statique – Déplacements

Si nous reprenons notre codage de la topologie, l'état du jeu initial « *00* », où nous faisons abstraction de l'historique de l'appariement des joueurs et de la construction du plateau de jeu, cet état est défini par :

```
<go:game bhb:id="00">
    <go:go-ban>
        <go:stone><?model links?></go:stone>
        <?model all stones?>
    </go:go-ban>
    <go:black>
        <bhb:user login="tom" passwd="<?model SHA256 login + passwd?>"/>
    </go:black>
    <go:white>
        <bhb:user login="thierry" passwd="<?model SHA256 login + passwd?>"/>
    </go:white>
</go:game>
```

Si le choix de **go**:black se portait sur la première **go**:stone (dont la position est arbitraire, donc l'hypothèse est sans perte de généralité) alors nous aurions :

```
<go:game>
    <go:go-ban>
        <?model all stones?>
    </go:go-ban>
    <go:white>
        <bhb:user login="thierry" passwd="<?model SHA256 login + passwd?>"/>
    </go:white>
    <go:black>
        <bhb:user login="tom" passwd="<?model SHA256 login + passwd?>"/>
```

*<**go**:stone><?model links?></**go**:stone>*

*</**go**:black>*

*</**go**:game>*

Dans notre codage, un coup (sans **go**:capture) est donc la combinaison de deux déplacements :

1. la **go**:stone de l'ensemble des pierres non jouées vers l'ensemble des pierres de **go**:black, ce qui représente le geste concret de poser une pierre tout en conservant les informations topologiques associées à l'intersection.
2. le **go**:black remis à la fin, qui représente qu'il a réalisé une action et que c'est maintenant à blanc de jouer.

<!-- C'est le principe du jeu de go, chacun pose une pierre à tour de rôle pour occuper un espace doté d'une topologie régulière. -->

Nous devons donc répondre à deux questions :

1. **Comment écrire une réécriture composée de plusieurs réécritures ?**

   *Il suffit de convenir d'une balise de composition, nous avons choisi **bhb**:compose.*

2. **Comment écrire cette réécriture de déplacement ?**

   De la façon suivante :

   *<**bhb**:link <?model où faut-il amener le segment?>>*

   *<**bhb**:link <?model début du segment?>/>*

   *<**bhb**:link <?model fin du segment?>/>*

   *</**bhb**:link>*

   *<!-- Cette règle complexe à trois **bhb**:link est le principal résultat de la campagne de recherche menée sur la logistique en 2012 -->*

   *<!-- Plus complexe que la transaction monétaire entre pairs, à deux **bhb**:link. -->*

```
<!-- Plus complexe que l'écriture dans un log, à un bhb:link.  -->
```

Dans le cas du coup de **go**:black sur la **go**:stone arbitrairement en tête :

```
<bhb:compose bhb:id="01">
    <bhb:link append="00/go:black">
        <bhb:link before="00/go:go-ban/go:stone[1]"/>
        <bhb:link after="00/go:go-ban/go:stone[1]"/>
    </bhb:link>
<!-- Et dans le cas du coup du déplacement de go:black : -->
    <bhb:link append="00">
        <bhb:link before="00/go:black"/>
        <bhb:link after="00/go:black"/>
    </bhb:link>
</bhb:compose>
```

Pour élucider comment cela fonctionne, prenons le cas du coup de **go**:black sur la **go**:stone arbitrairement en tête. D'abord, la règle est insérée et les URL internes sont résolues par des symboles :

```
<go:game>
    <go:go-ban>
        <bhb:link bhb:symbol="start-of-segment"/>
        <go:stone><?model links?></go:stone>
        <bhb:link bhb:symbol="end-of-segment"/>
        <?model all stones?>
    </go:go-ban>
    <go:black>
        <bhb:user login="tom" passwd="<?model SHA256 login + passwd?>"/>
        <bhb:link>
            <bhb:link bhb:symbol="start-of-segment"/>
```

```
                <bhb:link bhb:symbol="end-of-segment"/>
            </bhb:link>
        </go:black>
        <go:white>
            <bhb:user login="thierry" passwd="<?model SHA256 login + passwd?>"/>
        </go:white>
    </go:game>
```

Ensuite les symboles sont unifiés, ce qui, sans trop rentrer dans la technique du double-parcours que nous explorerons avec toute le finesse requise pour le système TED présenté en annexe, demeure ici assez intuitif et se passe sans difficulté :

```
    <go:game>
        <go:go-ban>
            <bhb:link/>
            <bhb:link/>
            <?model all stones?>
        </go:go-ban>
        <go:black>
            <bhb:user login="tom" passwd="<?model SHA256 login + passwd?>"/>
            <bhb:link/>
            <go:stone><?model links?></go:stone>
            <bhb:link/>
        </go:black>
        <go:white>
            <bhb:user login="thierry" passwd="<?model SHA256 login + passwd?>"/>
        </go:white>
    </go:game>
    <!-- L'élégance, l'économie de moyens et le caractère universel de notre technique de
    réécriture donne sa fluidité à l'ensemble de l'édifice. -->
```

Nous avons vu comment « *ajouter* » (**bhb**:link *x 1*), comment « *déplacer* » (**bhb**:link *x 3*), il nous reste à élucider comment « *relier* » (**bhb**:link *x 2*). Cette capacité sera illustrée en annexe avec la crypto-monnaie TOY.

De le même façon, notre technique de réécriture concerne l'ensemble des

données semi-structurées, quelles soient structurées comme dans l'exemple retenu ici, ou les données non structurées, les textes simples, comme illustré en annexe avec le système TED.

Phase Dynamique – Contrôle

Nous avons maintenant tous les éléments, toutes les briques serions nous tenté d'écrire, pour passer une phase cruciale de la description de notre premier exemple, la phase dynamique.

Dans la phase dynamique, l'ensemble des briques possibles de continuation est contenue dans l'application et gardé par des événements. Pour reprendre notre exemple nous avons déjà associé l'instruction informelle ci-dessous :

> <?act **noir** pose une pierre en hoshi.?>

à la brique composée présentée plus haut :

> *<**bhb**:compose **bhb**:id="01">*
>
> > <?Cf page 11?>

```
</bhb:compose>
```

Cette association va se faire par l'utilisation d'une balise capable de proposer un « *réflexe* » associé à un événement dans l'application.

Dans le cas présent, avec la topologie des pierres « *sous* » le **go**:go-ban, il suffit de détailler la règle de dessin des « *pierres vides* » en leur « *attribuant* » un réflexe, sous la forme de la brique composée présentée plus haut.

```
<xsl:stylesheet>

    <xsl:variable name="game" select="/go:game"/>

    <xsl:variable name="active-side" select="$game/*[2]"/>

    <!-- le deuxième nœud du jeu est le camp actif -->

    <xsl:variable name="player" select="name($active-side)"/>

    <xsl:variable name="user" select="$active-side/bhb:user/@login"/>

    <?view dessiner les pierres posées selon leur position et leur camp, comme réalisé plus haut?>

    <xsl:template match="go:go-ban/go:stone">

        <?view dessiner une intersection vide selon sa position?>

        <xsl:if test="bhb:user()=$user">

            <on:click <?view et lui attribuer, si l'intersection est simple?>>

                <xsl:variable name="root" select="bhb:identity($game)"/>

                <bhb:compose>

                    <!-- Déplacement de la pierre -->

                    <bhb:link append="{$root}/{$player}">

                        <bhb:link before="{bhb:identity(.)}"/>

                        <bhb:link after="{bhb:identity(.)}"/>

                    </bhb:link>

                    <!-- Déplacement du joueur (round robin) -->

                    <bhb:link append="{$root}">

                        <bhb:link before="{$root}/{$player}"/>
```

```
                    <bhb:link after="{$root}/{$player}"/>
                </bhb:link>
            </bhb:compose>
        <on:click>
    <xsl:if>
</xsl:template>
</xsl:stylesheet>
```

Les deux opérateurs clés introduit dans cet exemple concernent la notion d'identité :

- **bhb**:identity(xPath) est un **fonction** donnant l'identité du nœud désigné par le xPath. Cette identité est sous la forme du nom de la brique *où ce nœud a été introduit* et le chemin qui y mène *dans cette brique*. La brique en question peut être *arbitrairement distante* dans le temps.
- **bhb**:identity est une **balise** donnant l'identité de l'utilisateur dont la vue est calculée. Cette identité est unique.

```
<!-- associée à un mot de passe dans l'implémentation de l'identité retenue à titre illustratif. -->
```

Phase Dynamique – Topologie

Nous allons à présent traiter de la question de la go:capture, qui est quand même une notion avec un niveau d'expressivité important : « une go:stone x est vivante si et seulement si x peut être reliée par un chemin de go:stone du même camp vers un emplacement vide ».

```
alive(go:black) = lx : x/(go:link [ go:stone/go:black])*/go:stone/go:go-ban
```

Ce type de prédicats topologiques basé sur des « expressions régulières de chemin » n'est pas pris en charge par la syntaxe xPath. Pour pouvoir la décrire, nous demandons au développeur de nous ajouter la fonction bhb:path permettant de parcourir la topologie selon un chemin sans cycle et où chaque pas vérifie un critère de validité :

```
bhb:path('go:link','go:stone/go:black')/go:stone/go:go-ban
```

Pour déterminer les captures, il suffit d'insérer après la déclaration de la variable « player », les variables suivantes :

```
<!-- ici nous sommes presque en train de faire de la phase physique -->

<xsl:variable name="self" select="id(.)"/>

<xsl:variable name="capture"

select="bhb:path('go:link','go:stone/{$player}')/go:stone[ not(id(.)=$self]/go:go-ban"/>
```

Il faut aussi vérifier, en l'absence de capture que la pierre posée est vivante. Puis, pour chacune des captures, il faut replacer la pierre dans le pot commun, et conserver le compte des pierres capturés. A la fin de la partie, les emplacements vides qui ne peuvent pas rejoindre la couleur adverse sont comptés avec les capturés pour réaliser le score final et déclarer la **go**:victoire.

Phase Dynamique – Consensus

Le consensus intervient quand plusieurs machines différentes coopèrent pour élaborer le contenu de la chaîne de bloc. Dans le contexte des crypto-monnaies, qui s'étend ici au jeu de go, le principal objet du consensus est la détection et la répression des fraudes.

```
<!-- La notion de consensus sera élargie (Cf. Annexe TED). -->
```

Penser une machine qui fraude est étrange, tant la fraude parait liée à la notion de libre-arbitre. En effet, une machine peut être *byzantine*, c'est à dire programmée pour se comporter (presque) normalement, sauf qu'elle cherche à tromper les autres machines pour le bénéfice du fraudeur – qui demeure le programmeur, un humain, siège du libre-arbitre. Au go, ce genre de fraude serait de jouer deux pierres, ou d'en capturant de manière fantaisiste. Il est donc vital que les autres machines puissent vérifier toute contribution. C'est pour cela qu'il est essentiel qu'elle soit signée.

Avec cette signature, toute machine peut vérifier que l'action proposée est valide :

- selon sa propre application,

- dans la perspective du signataire.

<!-- En fait, le cœur de la contribution de M.Nakamoto est de dire que si la signature est infalsifiable, la majorité simple des machines arrivant un consensus disqualifient tous les comportements byzantins. Au bout du bout du raisonnement se niche le fait majoritaire, cela peut paraître spécieux, mais c'est en réalité très (très) solide. -->

<!-- L'autre condition clé est que le calcul du consensus soit obligé de terminer. Cette garantie est fourni par les calculs XSL – tant que chaque fonction le garantit, ce qui est du ressort des développeurs. XSL est donc un excellent (excellent) candidat pour être le langage de consensus. Et en plus des briques de continuation – utiles au consensus, XSL est parfaitement adapté pour générer tout le décor autour.-->

En fait, dans notre cadre, elle vérifie que le coup proposé est bien valide en regard des règles du jeu de go. Cela paraît évident pour un jeu à information complète, mais cela implique que toute blockchain soit un jeu à information complète. Ce qui est le cas.

*<!-- La question du **partage partiel** et des interfaces garantissant une forme de propriété privée collective entre les blockchain n'est pas traité dans ce document mais nous avons des arguments et même un [BREVET] sur cette question. Même si le mot blockchain m'était inconnu le 6 décembre 2010, le brevet est exprimé dans le cadre réparti d'un «* comportement transactionnel irréversible *» appelé «* apprentissage *». L'idée que tout **test** est la continuation d'une histoire sous forme de séquence inviolable d'atomes contributifs y est, par exemple, clairement exprimée. -->*

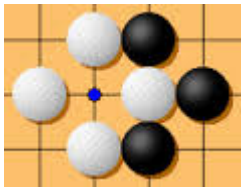
Nous définissons pour la validité d'une transaction le mode ***bhb**:compliant*, l'application de ce mode permet de construire une condition supplémentaire de validité à toute brique.

<!-- Nous verrons que pour la crypto-monnaie, ce calcul est un nombre (flottant) qui correspond au solde de l'utilisateur, devant être toujours positif. Ce calcul est réalisé après le calcul de l'XSL à réception de la brique finale (Cf. Annexe TOY)-->

*<!-- Il ne nous reste plus qu'un dernier élément métier a implémenter, le **go**:ko. Pour calculer cette condition, il faut exprimer l'idée que l'état visible du **go**:ban ne l'a jamais été dans le passé. →*

Le **go**:ko, l'impossibilité d'atteindre un état du passé, est le critère de validité parmi les plus complexes connu, sachant que le jeu de go est mentionné dans les écrits de Confucius, il y a plus de 2500 ans, cela ne lasse pas d'étonner. Sachant

que le même critère est utilisé dans les applications de model-checking[EVENTS] pour la validation des systèmes répartis, où encore que son analogie avec le second principe de la thermodynamique est patent, il est suffisamment générique pour que nous fassions l'effort de l'inclure.



Nous introduisons le symbole fonctionnel **bhb**:past() qui permet de récupérer toutes les « *versions linéaires* » d'un nœud depuis sa création.

```
<xsl:template select="/" mode="bhb:compliant">
    <xsl:variable name="all-states"
        select="bhb:compliant(bhb:past(go:game))"/>
    <xsl:value-of select="not(bhb:compliant(go:game)=$all-states"/>
</xsl:template>
<!-- On réduit chaque état à une chaîne de caractère pour avoir l'égalité syntaxique. -->
<xsl:template select="go:game" mode="bhb:compliant">
    <xsl:apply-templates select="descendant::go:stone">
        <xsl:sort <?model ordre lexicographique?>/>
        <!-- il doit exister des coordonnées pour dessiner les pierres -->
    </xsl:apply-templates>
</xsl:template>
<xsl:template select="go:go-ban/go:stone" mode="bhb:compliant">+</xsl:template>
<xsl:template select="go:black/go:stone" mode="bhb:compliant">0</xsl:template>
<xsl:template select="go:white/go:stone" mode="bhb:compliant">1</xsl:template>
```

La vérification par consensus de toutes les briques entre leur élaboration et leur scellement fait partie des fondamentaux du système. Ceci sera utile pour toutes les transactions monétaires où il faut s'assurer que le payeur dispose de la somme qu'il s'engage à régler et qui ne peut être déterminée à l'avance.

Nous avons implémenté la règle du **go**:ko. Mais si nous en restions là, le joueur pourrait élaborer un **go**:ko mais au moment du consensus sa continuation serait refusée – et la fenêtre de dialogue système lui donnerait un message d'alerte, alors qu'il s'agit d'un règle métier.

Pour coller au comportement souhaité nous pouvons remarquer que si un brique proposée à l'utilisateur n'est pas paramétrée par des entrées de l'interface, la vérification peut être fait avant l'envoi des continuations à l'utilisateur.

C'est le cas pour le go où chaque pierre jouable est associé une continuation spécifique pouvant être validée avant la soumission à l'utilisateur. Nous obtenons là *exactement* le comportement souhaité.

Phase Dynamique – Initiation

Nous avons implémenté l'ensemble du jeu de go, ne reste plus qu'à initier la partie ! Comme la règle du soupir le stipule, attention aux débuts ! En l'espèce, nous avons affaire à la problématique-clé des réseaux sociaux, l'appariement des personnes partageant une intention. Nous allons donc considérer un système simplifié d'appariement ce que nous avions fait, avec sagesse, dès le début du story-board :

> <?act tom initie une **partie de jeu de go** en 19x19.?>
>
> <?act tom rejoint la **partie** comme **noir**.?>
>
> <?act thierry rejoint la **partie** comme blanc.?>

Il reste quand même à résoudre une petite question : **comment initier une partie de go ?**

> *<!-- Si cette petite blague vous a donné un instant de vertige, c'est bon signe. C'est le vertige du soupir. Cela doit devenir une authentique sensation de danger : le passé est infini et il faut des cartes pour s'y rendre sans se perdre. -->*

Dans notre conception, une partie de jeu de go est l'application permettant de jouer une partie de go, donc la feuille de style étendue (XSL) qui fournit tout le décor (HTML, CSS) et toutes les continuations (BHB) nécessaires.

Initier une partie de go, c'est créer la blockchain hypertexte suivante sur son serveur avec ceci comme préfixe à son avenir :

```
<?xml version='1.0' encoding='iso-8859-1'?>

<?xml-stylesheet type='text/xsl' href='XSL'?>

<?xml-navigator href='NAV'?>

<bhb:user xmlns:bhb="ptp://hypertext.blockchain"
        login="tom" passwd="<?model SHA256 login + passwd?>"
        size="19"/>
```

Le serveur reconnaît ce fichier XML comme initiation et tente de résoudre l'URL de l'application (XSL).

> *<!-- Tous les serveurs de blockchain hypertexte sont des serveurs HTML et il serait tentant d'utiliser ce biais pour nommer les ressources utiles. Cela peut être fait dans un premier temps, pendant la montée en puissance, mais à (court-)terme, l'application (XSL) et tous ces composants (CSS, JS, …) seront eux-mêmes issus de blockchain hypertextes et l'ensemble du système dispose de son propre système de communication sur TCP/IP, avec son propre serveur de nom, le PTP : « perimeter transfert protocol ». En effet, à la différence du DNS classique, l'objet de ce serveur de nom est de vous donner accès à* ***un serveur****, parmi une multitude, de la blockchain de votre choix.*
>
> *<!-- si ce serveur venait à défaillir, il vous met en relation avec un autre, même vos données « éthérées », les briques que vous avez élaborées et qui ne font pas encore consensus, sont (en général) préservées. Nous pouvons émuler ce comportement avec des interfaces de récupérations qui s'activent si la connexion TCP s'écroule et proposent des reconnexions. →*

Soupir, le serveur de la partie a résolu l'URL, le serveur scelle le préfixe et démarre un instance de NAV sur une URL http « *propre à ce serveur* » et c'est parti. Le navigateur consulte le serveur qui lui demande son login et son mot de passe puis une fois qu'il a vérifié son identité, fait tourner l'application qui applique alors son gabarit d'initiation :

```
<xsl:template match="/bhb:user">
    <html:html>

        <on:load>

            <go:game>

                <?model go-ban @size x @size ?>

                <go:black>
```

```
                    <bhb:link before="bhb:identity(.)"/>
                    <bhb:link after="bhb:identity(.)"/>
                </go:black>
                <go:invite/>
            </go:game>
        <on:load>
    </html:html>
</xsl:template>
```

Ceci réalise le deuxième temps du story-board. Pour inviter son opposant, Tom diffuse l'adresse http locale sur son réseau social préféré, soupir. Thierry la clique, une fenêtre de son navigateur s'ouvre, lui est demandé son login et mot de passe, il soumet son identité et là...

```
<xsl:template match="go:invite">
    <xsl:choose>
        <xsl:when test="bhb:user()=../go:black/bhb:user/@login">
            <?view appel à la patience?>
        </xsl:when>
        <xsl:otherwise>
            <on:load>
                <bhb:link before="bhb:identity(.)"/>
                    <go:white>
                        <bhb:user login="{bhb:user()}"
                            passwd="{bhb:user('SHA256')}">
                    </go:white>
                <bhb:link after="bhb:identity(.)"/>
            </on:load>
        </xsl:otherwise>
    </xsl:choose>
```

```
</xsl:template>
```

Pour conclure, un petit peu de magie blockchain hypertexte. Ici l'ingénieur est certes pas une tanche en XSL, mais il a travaillé sans se préoccuper des problèmes de décentralisation et le consensus blockchain hypertexte va traiter pour lui, tout seul, le problème épineux de l'implosion de la source.

En effet, Tom a diffusé « *l'adresse http locale sur son réseau social préféré* » ! Et il a des milliers de followers mordu de go, qui cliquent comme des criquets dès que la notification apparaît. La source est submergée de requête et va-elle se prendre les pieds dans le tapis ?

En fait, la plupart des gens prennent ici le problème à l'envers. Rien ne peut empêcher le serveur d'autoriser à différent utilisateurs d'entrer, mais quand , en tant que serveur blockchain, il devra arriver à un consensus, peu importe qu'il soit seul ou entouré, dès lors qu'une continuation sera promue par la majorité, elle inhibera toutes les autres continuations, qui recevront le message système « *votre continuation n'a pas été acceptée.* ».

En s'appuyant sur ce savoir-faire, l'ingénieur a résolu sans le savoir un problème dont il ignore l'existence qu'il aurait très sûrement échoué à résoudre.

CQFD

Conclusion

La **méthodologie BHB** permet à des ingénieurs métiers *sans aucune notion en cryptographie répartie* de produire des applications générant des blockchain hypertextes avec toutes les bonnes propriétés. Comme nous venons de la voir, un ingénieur peut résoudre des problèmes répartis complexes sans même vraiment d'en apercevoir.

Pour ces ingénieurs, disposés à apprendre un peu d'hypertexte, la **blockchain hypertexte** se résumera alors à une réinterprétation de la notion du temps à la fois dans le « *processus implémenté* », appelé le **test** et dans le « *processus de son implémentation* », appelé le **code**. Notre méthodologie tente de définir un rapport géographique au temps dans le test et dans le code.

Une des idées centrale de cette méthodologie est de ne pas trop se préoccuper du passé, car le passé est une affaire de professionnel. Le but de l'ingénieur doit être d'objectiver l'intention, c'est à dire proposer à « *ses* » utilisateurs les continuations qui correspondent à leur besoins, ou à leur désir, leur intention.

Annexe TOY : Crypto-monnaie (2009)

<?TBD Les crypto-monnaies sont les premières applications des blockchain, un livre de compte est depuis toujours un registre de déclarations « éternelles ». ?>

Annexe MIRZA : Tableur coopératif (2011)

<?TBD La première application intégralement conçue selon les principes dirigeant l'ensemble de notre travail depuis 2001 ?>

Annexe TED : Coproduction de textes (2015)

<?TBD Comment réaliser un éditeur TED, ou Transfert Editor, remplaçant le copier-coller par une opération de « transclusion »[XANADU] ?>

<?TBD La question de l'écriture de la loi, ou plus généralement de tout règlement, contrat ou application, est la suite logique de ce travail et donne à l'ensemble une consistance industrielle ?>

Lexique des QName BHB

- **bhb**:block – **Balise** : regroupe des briques, contient les informations de provenance, non détaillée dans le présent document (TBD).
- **bhb**:compliant – **Mode** : Mode permettant de définir un état simplifié pour la validation. des briques
- **bhb**:compliant(_) – **Fonction xPath** : applique le mode ci-dessus.
- **bhb**:compose – **Balise** : permettant de mettre plusieurs réécriture dans la même brique.
- **bhb**:link – **Balise** : constructeur du mécanisme de réécriture.
- **bhb**:mask – **Balise** : mécanisme d'urgence, non détaillé dans le présent document (TBD).
- **bhb**:past(_) – **Fonction xPath** : fonction de parcours temporel.
- **bhb**:path(_,_) – **Fonction xPath** : fonction de parcours topologique.
- **bhb**:symbol – **Balise** : constructeur de l'extension de l'hypertexte au réseau.
- **bhb**:user() – **Fonction xPath** : fonction permettant de récupérer le nom de l'utilisateur courant et sa clé d'identification.

Bibliographie

[XANADU] **Getting it out of our system**

Ted Nelson (1965)

<?TBD **Système hypertexte à base de briques** (vu cet été au Science Museum)

Ted Nelson – (1973) ?>

[EVENTS] **Event Structures**

Glynn Winskel (1986)

[WWW] **Information Managament : A proposal**

Tim Bernes-Lee (1989)

[BREVET] **Procédé de partage d'apprentissage partiel d'une application logicielle**

Pierre Gradit (2010)

[BLOCKCHAIN] **Qu'est-ce que la blockchain ?**

https://blockchainfrance.net/decouvrir-la-blockchain/c-est-quoi-la-blockchain

[MODAL] **Une représentation innovante de la connaissance pour réaliser les systèmes d'information du futur**

Pierre Gradit (2015)