Un système de gestion de connaissances

Thibault Polge[1]

1. Doctorant contractuel chargé d'enseignement en philosophie (Centre de philosophie contemporaine de la Sorbonne, Université Paris 1 Panthéon Sorbonne). thibault.polge@univ-paris1.fr

Introduction

Ce document présente les fonctions fondamentales d'un outil d'organisation de données en recherche en histoire. Ce logiciel, en cours de conception, est écrit dans le cadre de ma thèse, c'est-à-dire qu'il vise à couvrir tout particulièrement les besoins d'un historien des sciences qui travaille sur la période moderne.

Tkacz veut modéliser les différentes activités d'organisation des données trouvées en bibliothèque ou aux archives. Il présuppose que ces activités consistent principalement en la prise en notes d'informations trouvées sur des sources papier ou numérisées. Il suppose que la prise de notes sur des livres ou des documents d'archives s'organise selon un flux de travail à (au moins) deux temps : 1) La prise rapide de notes peu structurées, directement à la lecture du document. Ces notes peuvent porter sur le document lui-même s'il constitue une source primaire, mais aussi potentiellement renvoyer vers d'autres sources à consulter, apporter des informations sur un objet tiers (une personne, un évènement, un autre document. . .), *etc*. Elles peuvent donc prendre différentes formes, comme une référence bibliographique, une citation, un évènement ou toute autre forme. Ensuite, 2) l'organisation et la mise au propre des points importants, et notamment la recopie de certaines informations sur des fiches spécifiques. Ces informations peuvent être une liste de sources à consulter citées ou évoquées dans le document, des données nouvelles sur une personne, etc.

Tkacz a pour objectifs de faciliter la manipulation de données sémantiquement structurées, sans que cette structure ne soit un obstacle à l'enregistrement d'informations complexes ; de réduire la nécessité d'une reprise (temps 2) des informations acquises sur un document sur de nouveaux supports, en en facilitant l'organisation et le tri ; de maintenir « vivant » le graphe des relations entre informations, autrement dit de ne pas disposer que de fiches isolées mais toujours un « répertoire » vivant et organisé.

Chapitre 1

Principes fondamentaux

Tkacz manipule trois types d'entités fondamentaux : des *fiches*, liées entre elles par un mécanisme de *relations*, et organisées dans des *taxinomies*. Tkacz manipule ces données sous la forme d'un graphe.

1.1 Fiches

La fiche est l'unité fondamentale de Tkacz. Une fiche peut représenter n'importe quoi, bien qu'elle soit pensée pour modéliser une (et une seule) entité objective. Une fiche a un *type*, qui est le type d'entité qu'elle représente : : une personne, une publication, un évènement, *etc*. Ces types sont définis par le schéma en cours d'utilisation (cf. chapitre 3). Une fiche contient deux grandes séries de données :

1. des métadonnées structurées qui décrivent l'entité qu'elle représente (dans le cas d'une fiche personne, il s'agira par exemple de son nom, son prénom, sa date de naissance, *etc*.). Ces métadonnées peuvent être obtenue automatiquement dans certains cas : l'importation d'une référence bibliographique peut se faire depuis une base de données externe, un script étant chargé de la conversion.
   La structure des métadonnées fait que la plupart des fiches n'ont pas de titre explicitement attribué : celui-ci est calculé à partir des métadonnées (une fiche représentant une personne ou une publication n'a pas besoin de titre : le nom de cette personne ou le titre, les auteurs et la date de cette publication représentent son contenu). Ainsi, une fiche d'entrée bibliographique produit son titre automatiquement (mais de façon configurable) par rapport aux données structurées qu'elle contient. Un tel titre est uniquement destiné à rendre la fiche aisément identifiable par l'utilisateur, en interne, il lui est attribué un identifiant numérique.
2. des notes de forme libre. Il peut y avoir autant de notes que nécessaires sur une fiche. Ces notes peuvent elles-mêmes décrire des métadonnées (voir plus bas).

Il est possible de lier à une fiche des fichiers externes (par exemple, pour un article décrit dans une fiche référence bibliographique, l'artice lui-même en PDF). Les associations à

ces fichiers sont gérées comme des métadonnées, et les fichiers sont stockés dans le dépôt Tkacz.

Fragments

Les *fragments* sont des fiches qui peuvent être incluses dans les notes d'autres fiches, et ne sont pas *a priori* des fiches autonomes. Les images ou les citations sont des fragments. Ils se distinguent des fiches ordinaires par quelques propriétés :

1. Ils sont généralement créés à la volée en prenant des notes : inclure une image ou copier une citation, c'est créér un fragment.
2. Ils ont obligatoirement un contenu. Contrairement à une fiche qui peut se contenter de faire référence à une entité, un fragment la contient. Il peut exister un type de fiche tableau qui décrit un tableau, mais un fragment de type image n'a de sens que s'il contient effectivement une image.
3. Ils peuvent être intrinsèquement liés à (au moins) une fiche : par exemple, une citation n'est jamais déconnectée de la référence dont elle est extraite. Ils peuvent être liés à plusieurs, comme dans le cas d'une citation qui apparaît à l'identique dans plusieurs éditions d'un même texte. Ils peuvent aussi être autonome (une image n'est pas forcément liée à une fiche).
4. Ils peuvent être inclus dans d'autres fiches que celle de laquelle ils dépendent, sans perdre leur lien initial (une citation peut être citée ailleurs).
5. Comme une fiche ordinaire, ils peuvent être organisés dans des taxinomies (cf. section 1.3).
6. Ils n'apparaissent pas dans les vues standards.

1.2 Relations

Une fonction fondamentale et originale de Tkacz est sa capacité à décrire des relations complexes entre des fiches, des métadonnées et d'autres entités. Ces relations sont sémantiques, et font d'un dépôt Tkacz une sorte de graphe. Les relations entre fiches ont quelques propriétés importantes :

1. Elles peuvent être fixées dans les métadonnées (attribution d'un auteur, par exemple) ou directement dans les notes. Ce ne sont généralement pas les mêmes relations qui sont décrites de l'une ou de l'autre façon.
2. Elles sont nettement plus complexes qu'une simple relation au sens de ce terme dans un SGBDR. Une relation contient elle-même des métadonnées, et elle peut être annotée. Au sens strict, *une relation est elle-même une fiche d'un type particulier*.

Les relations les plus simples sont fixées dans les métadonnées : par exemple, l'auteur d'une publication n'est pas une chaîne de caractères, comme « W. V. O. Quine » : c'est une personne, qui est donc représentée par sa propre fiche. La métadonnée « auteur » est donc une relation vers une (ou des) fiches de type « personne ». S'il n'existe pas de fiche pour cet auteur, elle est créée à la volée.

Mais même une relation comme « être auteur de » peut-être plus complexe qu'un lien simple entre deux fiches, comme on pourrait le concevoir dans un SGBDR : il n'est pas rare par exemple de trouver une publication sous pseudonyme, ou dont l'attribution est douteuse. La fiche cible peut bien contenir le(s) pseudonyme(s) en plus du nom légal (ou *des* noms légaux). Mais c'est le modèle de la relation « auteur » qui prévoit qu'elle puisse non seulement pointer sur la fiche cible, mais aussi préciser quel nom est utilisé (ou quelle graphie, ou même avec quelle faute il est copié). Ce modèle permet aussi de préciser que l'attribution est douteuse, ou bien fausse. On peut même modéliser une situation telle que « publié sous le pseudonyme qui peut correspondre à telle personne ou telle autre personne »). Dans la mesure où le modèle de données de Tkacz est extensible par l'utilisateur (cf. chapitre 3), une relation permet en fait de préciser potentiellement n'importe quoi.

Qu'une relation soit une fiche est ici un avantage considérable : il peut être très utile, dans les cas d'attribution douteuse par exemple, de prendre des notes sur l'attribution elle-même (qui prétend que c'est *x*, qui prétend que c'est *y*, *etc.*)

Une relation peut aussi exister entre des données qui ne sont pas des fiches. Différents noms d'une même personne peuvent être en relation l'un avec l'autre : « FBI » est l'acronyme de « Federal Bureau of Investigation » (et par corollaire, celui-ci est la forme développée de celui-là). Ce type de relation permet, lors de l'emploi des références comme citations dans un document (par exemple en exportant la bibliographie en BibTEX), d'attribuer convenablement les champs `Author` et `Shortauthor`.

Corollaires

Une relation peut avoir un *corollaire*, c'est-à-dire que la relation de A et B implique une relation (de même nature ou d'une autre nature) de B vers A. Dans l'exemple de l'auteur, le lien « a pour auteur » a pour corollaire « est auteur de ». Un corollaire est toujours attribué automatiquement, puisqu'il est une conséquence logique de la relation dont il est le corollaire : $Ax, y \iff Bx, y$.

Une relation est un pur corollaire lorsqu'elle ne peut pas être assignée directement : « est l'auteur de » est un exemple de pur corollaire, qui ne peut être affecté à une fiche personne, mais dérive de l'affectation de la relation « auteur » d'une fiche document à une fiche personne.

Une relation dont le corollaire est la même relation de la cible vers la source est dite *réciproque* ou *symétrique* : $Ax, y \iff By, x$

1.3 Taxinomies

Cette organisation se fait sous la forme de **taxinomies**. Une taxinomie est une structure hiérarchique, comparable à un système de fichiers, dans les entrées duquel les fiches prennent place. Contrairement à un système de fichiers, par définition unique, plusieurs taxinomies peuvent cohabiter, aucune fiche ne réside nécessairement dans une taxinomie quelconque, et une fiche peut se trouver associée à plusieurs taxons.

Une taxinomie peut être créee manuellement ou automatiquement. Les types de fiche sont eux-mêmes des taxons, et créent une taxinomie automatique et non modifiable (ie, associer une fiche à un type revient à la faire rentrer dans une taxinomie)

Construction des taxinomies

La création des taxinomies peut être automatique ou manuelle. La création automatique se fait par du code Python fourni par le schéma ou l'utilisateur, la création automatique est faite par l'utilisateur.

Affectations des fiches aux taxons

Principes généraux

L'affectation peut-être manuelle (chaque fiche est ajoutée par l'utilisateur), automatique (le taxon contient certaines fiches selon certaines propriétés) ou hybride : l'affectation est automatique par défaut, mais peut être forcée. Par exemple, une taxonomie des acteurs d'un champ par activité professionnelle privilégiera une affection manuelle des fiches ; une taxonomie des publications par décennie sera automatique et l'affectation se fera en fonction de la première publication.

Les modes d'affectation automatique sont les suivants

Relations entre taxons

Les taxons peuvent se voir associés un certain nombre de règles d'affectation :

— un taxon peut contenir la totalité du contenu (ie, les fiches) de ses enfants. Comme dans une taxinomie biologique, toutes les sous-classes de mammifères (theria et prototheria) *sont* des mammifères ; ou au contraire ne contenir que ce qui y est explicitement ajouté.

— Un taxon peut être incompatible avec un autre, ie la présence d'une fiche dans ce taxon rend impossible sa présence dans un autre. Par exemple, un mammifère ne peut pas être un poisson ; ou au contraire un taxon peut en impliquer automatiquement un ou plusieurs autres.

Un taxon peut fixer des règles pour lui-même et/ou ses enfants à un niveau n ou aux niveaux n à m.

Le lien d'une fiche à un taxon est lui-même une fiche, qui peut donc être commenté.

Le fait qu'une taxinomie est forcément hiérarchique n'implique pas nécessairement qu'elle soit manipulée comme telle. Il est possible de créer des taxinomies de « tags » ou tous les tags sont au même niveau.

Les taxinomies ne sont pas fortement indépendantes ; elles sont gérées en interne comme un unique arbre hiérarchique.

Première partie

Utilisation

Chapitre 2

Interface utilisateur

L'interface principale de Tkacz est une interface graphique conçue avec Qt, et conçue pour fonctionner indifféremment sous Linux, OS X et Windows.

Tkacz est accessible en ligne de commande via la commande `tkacz` (et éventuellement sa version abrégée `tz`). Cet exécutable unique permet d'invoquer des commandes secondaires, à la manière de Git ou Aptitude. Ces sous-commandes prennent généralement une forme sujet-verbe [complément], par exemple :

```
$ tkacz card edit Foucault
# |     |    |    |
# \_ Exécutable   |
#       |    |    |
#       \_ sujet  |
#            |    |
#            \_ verbe
#                 |
#                 \_ complément
```

Chapitre 3

Schémas et modèles de données

Tkacz est conçu selon un modèle de données à deux niveaux : le premier niveau est celui du noyau du logiciel, décrit chapitre 1. Ce noyau, qui connaît les notions de fiches, taxinomies et relations, est encore insuffisant pour les manipuler effectivement : des modèles de données, extérieurs, décrivent les types de fiches, les taxinomies où elles peuvent entrer et les relations possibles entre elles. Ainsi, c'est le modèle standard, et pas le noyau Tkacz, qui définit les types de fiches « personne » ou « publication ».

Du point de vue de l'implémentation, cette distinction est faite de la façon suivante : le noyau du logiciel (écrit en C++) charge des types de données décrits en XML [1] et Python [2] qui représentent la forme générale des données (la forme abstraite des fiches, des taxinomies et des relations), ainsi que leur représentation dans l'interface utilisateur (présentée chapitre 4). Le modèle de données standard décrit par exemple les types de fiches « personne » (physique ou morale), ou « publication ».

Cette conception duelle de Tkacz répond à l'impératif fondamental de ne pas figer le modèle de données : si Tkacz fournit bien un modèle standard, l'utilisateur peut l'étendre à son gré, le modifier, voire en écrire un nouveau [3].

Les modèles de données décrivent :

1. Des types de données, qui complètent et étendent les types natifs de Tkacz.
2. Des types de fiches.
3. Des types de relations entre ces fiches.

1. Rien n'interdit par ailleurs d'utiliser un autre langage de balisage, mais l'implémentation initiale se restreint à XML.

2. Le choix de Python est principalement dû à deux considérations : ce langage est largement répandu, facile à utiliser et à apprendre ; il dispose de nombreux outils — y compris des notations — pour manipuler des ensembles ; il est très facile à interfacer sur du code C/C++. L'implémentation actuelle du système de modèles fait appel à certaines structures très spécifiques de Python (notamment au niveau de la POO), mais pour autant, la dépendance à ce langage précis n'est pas un impératif, et il n'est pas impossible que d'autres soient interpréteurs soient joints à Tkacz.

3. L'interface graphique sera sans doute conçue en fonction du modèle standard. Dans ce cas, la conception de nouveaux schémas pourra impliquer de réécrire une partie du code d'interface.

4. Des schémas, composés d'un ensemble de types de données, de fiches et de relations.
5. Des descriptions abstraites d'interface utilisateur pour modifier certains types de données.

Ce chapitre présentera les quatre premiers points, le chapitre 4 est consacré à la description des interfaces.

3.1 Notions fondamentales

Les types de données, de fiches, de relations ou de taxinomies sont des descriptions XML qui peuvent être étendues par du code Python. À un type correspondent généralement deux fichiers, un fichier `.xml` qui décrit la structure de données, et un `.py` qui contient le code exécutable.

L'utilisateur peut étendre le modèle de données en ajoutant des attributs ou du code. Utiliser deux langages distincts répond à un impératif de sécurité : dans la mesure où l'essentiel des extensions utilisateurs peuvent se représenter en XML, on évite de rajouter du code exécutable au sein des dépôts.

Par défaut, aucune taxinomie ni relation n'est disponible, mais Tkacz propose un certain nombre de types primitifs, liés au code du noyau, et qui ne peuvent pas être remplacés. Ces types sont :

Nom	Description
`array`	Tableau
`bool`	Booléen : vrai/faux.
`dict`	Tableau associatif
`enum`	Énumération
`number`	Nombre
`set`	Ensemble
`string`	Chaîne de caractères
`variant`	Type variable

TABLE 1: Liste des types primitifs avec leur description

3.2 Types de données

Le format des types de données est à peu près le même que celui des types de fiches ou de relations. Par conséquent, cette exposition va être relativement longue, mais couvrira l'essentiel de ce qui est nécessaire pour décrire n'importe quel modèle Tkacz.

Un type de données est en gros l'équivalent d'un `struct` en C (ou d'une classe, s'il contient du code). Tkacz gère l'héritage et le polymorphisme : un type de données peut en étendre un autre et un attribut peut être d'un type abstrait. Dans ce cas, les différents types non abstraits qui l'étendent sont proposés pour fixer la valeur de l'attribut.

Le modèle XML

La construction d'un type se fait en définissant des attributs typés

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <model id="myModel" version="1.0.0" xmlns="http://tkacz.thb.lt/ns/model/">
3
4      <datatype id="myType" />
5
6  </model>
```

Ce code définit un type `myType` vide. Le paramètre `id` définit le nom interne du type. Il est obligatoire. Le nom interne complet est constitué de la concaténation du nom du type et de la totalité des noms de ses parents : ici, il sera `myModel.myType`.

Ce type vide n'est pas très intéressant. Ajoutons-y des attributs :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <model id="myModel" version="1.0.0" xmlns="http://tkacz.thb.lt/ns/model/">
3
4      <datatype id="myType">
5          <attr type="string" id="myString" />
6          <attr type="number" id="myNumber" />
7      </datatype>
8
9  </model>
```

Les types standards disposent chacun d'un raccourci de création d'attribut. Ainsi, on peut aussi écrire :

```
5          <string id="myString" />
6          <number id="myNumber" />
```

Un type peut avoir, en plus de ses attributs, des *propriétés* : une propriété décrit le comportement de l'instance du type, mais n'est pas en soi une donnée sémantique. Un

nom de personne, par exemple, peut être composé de plusieurs parties, c'est à dire d'un prénom, d'un nom de famille, éventuellement d'un second prénom, *etc.* ; ou bien être une unique séquence, comme certains noms non-occidentaux ou les noms des personnes morales. Qu'une fiche représente une personne physique ou une personne morale est un attribut de la fiche, mais pas de son attribut « nom ».

```
        <string id="myString" />
        <number id="myNumber" />

        <property type="string" id="myProperty" />
```

Les champs et les propriétés ne sont pas limités aux types standards, n'importe quel type personnalisé peut être utilisé. Créons un second type qui utilise le premier :

```
<?xml version="1.0" encoding="UTF-8"?>
<model id="myModel" version="1.0.0" xmlns="http://tkacz.thb.lt/ns/model/">

    <datatype id="myType">
        <string id="myString" />
        <number id="myNumber" />

        <property type="string" id="myProperty" />
    </datatype>

    <datatype id="myOtherType">
        <attr type="..myType" id="myTypeAttrA" />
        <attr type="..myType" id="myTypeAttrB" />

        <property type="string" id="myStringProperty" />
    </datatype>

</model>
```

Les deux points avant `myType` indiquent de chercher le type un niveau plus bas dans l'arbre hiérarchique. Comme `myOtherType` a pour identifiant complet `myModel.myOtherType`, `..myType` désigne donc `myModel.myType`. Un seul point aurait désigné `myModel.myOtherType.myType`. Sans point initial, le nom est supposé complet : il aurait alors fallu écrire `myModel.myType`.

Un type peut contenir un autre type : cela est utile si le type secondaire n'est utilisé que localement. Dans ce cas, on peut écrire :

```
<?xml version="1.0" encoding="UTF-8"?>
<model id="myModel" version="1.0.0" xmlns="http://tkacz.thb.lt/ns/model/">

    <datatype id="myOtherType">
        <datatype id="myInternalType">
            <string id="myString" />
```

```
            <number id="myNumber" />

            <property type="string" id="myProperty" />
        </datatype>

        <attr type=".myInternalType" id="myTypeAttrA" />
        <attr type=".myInternalType" id="myTypeAttrB" />

        <property type="string" id="myStringProperty" />
    </datatype>

</model>
```

Un type défini à l'intérieur d'un autre type peut être utilisé par n'importe quel autre type, y compris dans un autre modèle.

Si un type n'est utilisé qu'une seule fois, il n'y a même pas besoin de le définir à part ni de le nommer :

```
<?xml version="1.0" encoding="UTF-8"?>
<model id="myModel" version="1.0.0" xmlns="http://tkacz.thb.lt/ns/model/">

    <datatype id="myOtherType">
        <attr id="myTypeAttrA">
            <string id="myString" />
            <number id="myNumber" />

            <property type="string" id="myProperty" />
        </attr>

        <property type="string" id="myStringProperty" />
    </datatype>

</model>
```

De manière générale,

Chaque attribut contient quelques propriétés fondamentales et des évènements standards :

Propriété	Type	Description
`enabled`	Booléen	Détermine si l'attribut est activé.
`null`	Booléen	Détermine si l'attribut a une valeur

TABLE 2: Propriétés primitives des types de données

Évènement	Description
`onChanged`	La valeur de l'attribut est modifiée
`onEnabled`	L'attribut est activé
`onDisabled`	L'attribut est désactivé
`onCleared`	L'attribut est remis à zéro
`onSet`	L'attribut qui était `null` prend une valeur.

TABLE 3: Propriétés primitives des types de données

Extension des types primitifs

S'il étend un type primitif (par exemple `<datatype id="myStringType" extends="string" />`), il a en plus une valeur de base : il continue d'être une chaîne avec d'autres propriétés, ce qui permet d'écrire quelque chose comme :

```
1 myObject.myTypeInstance = "une chaîne"
2 myObject.myTypeInstance.myAttribute = true
```

Code Python

3.3 Types de fiches, types de relations, types de taxons

Décrire un type de fiche ou de relation est quasiment la même chose que décrire un type de données, à ceci près que l'élément racine ne sera pas `datatype` mais respectivement `cardtype` et `reltype`.

3.4 Exemples

La relation « attribution d'une source »

La relation d'attribution d'une source à une information se fait entre n'importe quoi et une publication. Elle contient une propriété supplémentaire, qui est un emplacement dans la publication. Cet emplacement peut s'exprimer de différentes manières : si en général il s'agit d'un numéro de page, d'une section (*L'archéologie du savoir*, III, II, d), d'une pagination spécifique (*Théétète*, 142c), ou de plusieurs paginations combinées (*Critique de la raison pure*, A20/B34)

Le type abstrait « référence bibliographique » prévoit un attribut pagination, qui décrit comment le document est paginé. Cet attribut est défini comme suit :

```
. . .
<attr id="pagination">
    <datatype extends="enum" id="sectioning">
        <super>
            <extensible>true</extensible>
            <ordered>true</ordered>
            <option id="volume" />
            <option id="tome" />
            <option id="book" />
            <option id="chapter" />
            <option id="section" />
            <option id="subsection" />
            <option id="subsubsection" />
            <option id="paragraph" />
            <option id="subparagraph" />
            <option id="subsubparagraph" />
        </super>
    </datatype>

    <datatype id="sectionFormatting">
        <attr type="..sectioning" id="level" />
        <attr type="core.numberFormat" id="format" />
        <python>
            @bind(".level.onChange")
            def disableFormat(target, *args, **kwargs):
                target.format.enabled = target.level != "unpaginated"
        </python>

    </datatype>

</attr>
myObject.myTypeInstance.myAttribute = true
. . .
```

Chapitre 4

Liens des modèles de données à l'interface d'utilisateur

Voici la description de l'interface de saisie d'un nom de personne :

```
<block>
<if propname="simple">
    <lineinput bind="name" />
<else>
    <lineinput bind="name" />
    <hr />
    <lineinput bind="prefix" class="small" />
    <lineinput bind="firstName" class="small" />
    <lineinput bind="middleName" class="small" />
    <lineinput bind="vonPart" class="small" />
    <lineinput bind="lastName" class="small" />
    <lineinput bind="suffix" class="small" />
</else></if>
</block>
<checkbox bind="simple" />
```

Deuxième partie

Implémentation

Chapitre 5

Format de stockage

Un dépôt Tkacz est un répertoire du système de fichiers, qui n'est pas destiné à être manipulé par l'utilisateur. Il peut être présenté comme un bundle (sur OS X) ou stocké zippé (sur les autres systèmes) pour empêcher toute manipulation destructrice.

Ce répertoire contient la totalité des ressources internes de Tkacz (les fiches, la structure, etc.) et les fichiers liés.

5.1 Format des fiches

Les fiches sont stockées comme des fichiers textes et organisées dans des dossiers correspondant à leurs types et à leur identifiant interne. La fiche 627 de type personne physique sera stockée dans `/cards/person/natural/627/`. Chaque dossier contient :

— Une description XML des métadonnées et des relations de la fiche (`record.xml`)
— Les différentes séries notes associées à la fiche ()

5.2 Format des dépôts

Un dépôt combine un dépôt git [1] et une base de données SQLite qui sert de cache. Un dépôt vide a donc la structure suivante :

1. Dans ce document, le mot « dépôt » seul fait *toujours* référence à un dépôt Tkacz.

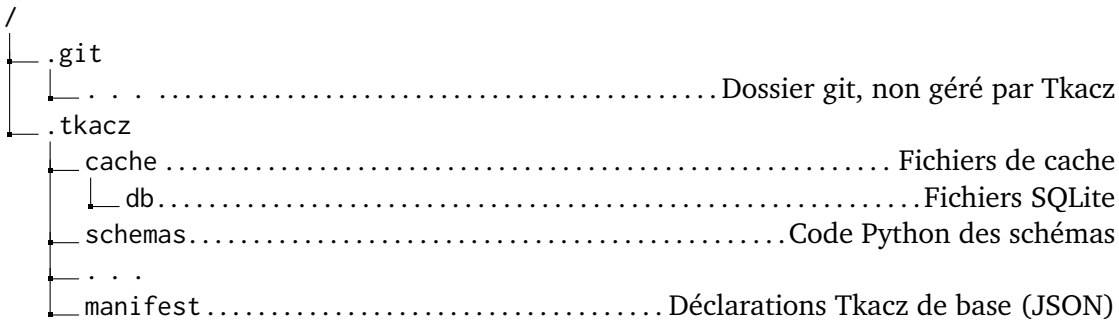

FIGURE 4 – Organisation des fichiers d'un dépôt Tkacz

Le fichier `.tkacz ▸ manifest` est une représentation XML de la structure détaillée ci-dessous.

```
{
    "tkacz": {
        "frameworkVersion": [0,1,0],
        "formatVersion":    [0,1,0],
        "schemaId":         "core",
        "schemaVersion":    [0,1,0]
    },
    "schemas": {
        "core.historian": {
            "version" :     [0,1,0]
        }
    },
    "repository": {
        "uuid":             "03095EEF-6C87-430B-A00E-440616196C31",
        "name":             "As set by the user"
    },
    "core": {}
}
```

Les clés `tkacz`, `repository` et `core` (options du schéma standard) et de façon générale toutes les clés racines vérifiant `[a-zA-Z]+[a-zA-Z0-9_]*` sont réservées, les extensions ou les schémas tiers peuvent inscrire leur paramétrage dans des clés au format `com.domaine.nom` (à la Java).

5.3 Stockage des fiches

Les fiches sont sauvegardées comme des fichiers gérés par Git, dans des dossiers correspondant à leurs types, par exemple `person ▸ collective`. Leur nom est un numéro attribué séquentiellement pour le dépôt entier (indépendemment du type donc). Tous les noms vérifiant `[0-9]+` sont donc réservés pour les fiches au niveau du suivi des versions.

Le dépôt n'utilise qu'une seule branche, `master`.
L'usage de Git fait ici est assez particulier : chaque message de commit décrit l'état des références, c'est à dire la fiche qui sert de source et la position dans cette fiche.
Quand Tkacz contrôle l'éditeur de texte, il commite automatiquement les changements, sans contrôle possible de l'utilisateur, dans les situations suivantes :

1. La saisie ou l'effacement se poursuit, mais la position du curseur a changé. [`move`]
2. L'utilisateur commence à saisir du texte après en avoir effacé. [`insert`]
3. L'utilisateur commence à effacer du texte après en avoir saisi. [`delete`]
4. La référence ou l'emplacement dans la référence a changé. [`insert`]
5. L'utilisateur coupe du texte. [`cut`]
6. L'utilisateur colle du texte (au format Tkacz, sinon `insert`). [`paste`]

Si un éditeur externe est utilisé, un commit est réalisé à chaque modification du fichier[2].

Format des messages de commit

Commits sur modification

Le format d'un message de commit en cours d'édition est une représentation JSON (minimisée) de la structure :

```
{
    "operation": "type",
    "using":     234,
    "position": {
        "page": "x"
    }
}
```

`operation` décrit l'action de l'utilisateur qui justifie le commit.
Avec cette réserve que le format de `position`, s'il est obligatoirement un dictionnaire, n'est pas déterminé par avance, et dépend du type de fiche. En imaginant des notes prises à la volée sur un enregistrement audio, `position` pourrait avoir un format du type :

```
{
    "position": {
        "tape": 0,
        "time": [0, 12, 34]
    }
}
```

2. Il est plausible qu'il soit en fait impossible d'utiliser un éditeur pour lequel Tkacz ne puisse pas suivre chaque modification. En effet, la fonctionnalité fondamentale de suivi des ajouts et modifications et de leur rattachement à des emplacements précis des sources nécessite un suivi extrêment précis des modifications. Autrement dit, seul peut sans doute être utilisé un éditeur qui sauvegarde à chaque caractère modifié.

Si une fiche est modifiée sans référence ouverte, le message de commit ne contient que la clé `operation`.

Commits vides

Des messages supplémentaires sont produits à l'ouverture et à la fermeture d'une ressource, avec des commit vide :

```
{
    "opening": 123
}
```

et :

```
{
    "closing": 123
}
```

Un dernier type de message permet d'associer une donnée déjà saisie à une autre ressource :

```
{
    "link": 72,
    "from": [132, 21],
    "to":   [135, 12]
}
```

`from` et `to` sont ici des paires ligne/colonne.

Calcul des diffs, suivi de l'origine et déplacement de paragraphes

Le *déplacement* de blocs de texte est un problème sérieux que les diff classiques ne permettent pas de résoudre. Or, un bloc déplacé doit continuer à être associé à son origine. Plusieurs solutions semblent possibles :

— Commit automatique à chaque opération couper/copier/coller.

— "Lester" les copies/coupes avec les informations d'origine de la source et les porter dans le commit lors du collage. *Cette option exclut absolument d'utiliser un autre éditeur de texte que celui de Tkacz*.

5.4 Structure du cache

La totalité des données utiles se trouve dans les fiches elle-même, ce qui signifie que seul le dépôt git doit être copié pour cloner entièrement

Chapitre 6

Communication avec Python

Tkacz est fondamentalement un framework Python : les fonctions de bas niveau sont implémentées en C++, mais les structures de données et les opérations sur les données sont écrites en Python.

Troisième partie

Annexes

Glossaire

corollaire @TODO.
    pur @TODO.

dépôt @TODO.

fiche @TODO.
fragment @TODO.

relation @TODO.
    réciproque @TODO.
    symétrique @TODO.

schéma Un schéma est un ensemble de type de données, de types de données et de relations qui forment un tout cohérent pour la manipulation de certains types de données.

Tableaux et figures

Table des matières