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

Chapitre 1

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 2

Principes fondamentaux

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

2.1 Fiches

La fiche est l'unité atomique de Tkacz. Une fiche peut représenter n'importe quoi, bien qu'elle représente en général 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). Chaque fiche a un *titre*, ainsi généralement que d'autres métadonnées, et peut contenir des *notes*. Les métadonnées exactes qui peuvent être contenues dans une fiche sont déterminées par son type.

Les *notes* liées à une fiche sont celles que prend l'utilisateur sur l'objet représenté par la fiche. Leur forme est libre, leur syntaxe est inspirée de Markdown. La syntaxe des notes est exposée chapitre 5.

Les métadonnées d'une fiche décrivent l'entité qu'elles représentent. Le titre est un de ces métadonnées, même si la plupart des types n'ont pas *stricto sensu* de titre[1] : 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 peut prendre la forme :

The efficacy of AZT in the treatment of patients with AIDS. . . 1987
Fischl, Richman, Grieco *et al*. N Engl J Med

1. Il existe un type « page blanche », qui a bien une métadonnée titre, et aucune autre. Mais ce type est une exception.

Automatiquement calculée à partir des noms des auteurs, de l'année et du titre (éventuellement, comme ici, abrégé) d'une entrée bibliographique. Le titre d'une fiche personne prendra une forme du type :

FOUCAULT, Michel 1926–1984

2.2 Relations

Une fonction fondamentale et originale de Tkacz est sa capacité à décrire des relations complexes entre des fiches. 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. La fiche cible peut bien contenir le(s) pseudonyme(s) en plus du nom légal (voire 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.*

est un attribut complexe, qui permet de lier des fiches entre elles de façon moins formelle et plus fine. Une relation a une **nature**, qui est l'équivalent du type d'une fiche. Dans certains cas, un lien peut être remplacé par une relation. Par exemple, dans un document

historique, l'attribution de l'auteur peut être douteuse — « auteur probable » est une relation, car le lien simple ne suffit pas — il faut sans doute préciser la nature du doute, les différentes sources, etc.

Une relation peut aussi décrire des relations entre entités : « membre de » ou « ami de », « frère de », etc.

Une relation peut être réciproque ou non. Dans l'exemple qui précède, les deux dernières sont nécessairement réciproques. Une relation peut aussi avoir un **corollaire**. La relation « A est membre de B » (dans cet exemple, B est par exemple une personne morale) a pour corollaire « B a comme membre A »

Une relation est de type « cite » ou « évoque » , qui permet de commenter un livre en le liant à ce qui fait son objet.

Formellement, une relation est toujours un prédicat à deux places et représente un nœud d'un graphe. Lorsque une fiche présente la même relation a plusieurs autres fiches, la relation est assignée autant de fois qu'il y a de fiches cibles.

Corollaires, corollaires purs et relations symétriques

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 comme une conséquence logique de la relation dont 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 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 dans l'autre sens est une relation symétrique.

2.3 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.

2.4 Synthèse

Tout est fiche.

Première partie

Utilisation

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 2. 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 ou de relations sont des objets Python qui étendent des classes de base de Tkacz. Ces trois types sont quasiment identiques : la différence entre eux est principalement dans la façon dont le cœur les manipule, pas dans leur structure. Leurs superclasses sont respectivement `TZDataType`, `TZCardType` et `TZRelationshipType`, dans le paquet `tkacz`.

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 :

TABLE 3.1: Liste des types primitifs avec leur description

Nom	Description
`TZBoolean`	Booléen : vrai/faux.
`TZDict`	Tableau associatif
`TZEnum`	Énumération
`TZFloat`	Nombre à virgule flottante
`TZInteger`	Nombre entier
`TZNumber`	Nombre
`TZSet`	Ensemble (éventuellement ordonné)
`TZString`	Chaîne de caractères
`TZVariant`	Type variable

3.2 Types de données

La description d'un type de données se fait en combinant et/ou en étendant des types natifs, sous la forme d'une classe qui étend `TZDataType` ou une métaclasse dérivée (*ie*, un autre type de données, y compris un type natif).

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

```
1 from tkacz import *
```

```

class MyDataType(TZDataType):

    myString = TZString()
    myNumber = TZNumber()
    myDate   = TZDate()
```

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 ».

Les propriétés se définissent en passant un type d'attribut au constructeur de `TZProperty`. L'exemple ci-dessous ajoute une propriété de type nombre :

```
from tkacz import *

class MyDataType(TZDataType):

    myString = TZString()
    myNumber = TZNumber()
    myDate   = TZDate()

    myProperty = TZProperty(TZNumber())
```

Chaque type (primitif ou dérivé) fournit des évènements, sur lesquels du code peut se brancher. On connecte des méthodes à des attributs ou des propriétés avec le décorateur `Hook` :

```
from tkacz import *

class MyDataType(TZDataType):

    myString = TZString(null=False)
    myNumber = TZNumber()
    myDate   = TZDate()

    myProperty = TZProperty(TZInteger())

    @Hook("myNumber.onChanged")
    def calculate(self):
        pass
```

Chaque attribut contient quelques propriétés fondamentales et des évènements standards :

TABLE 3.2: Propriétés primitives des types de données

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

TABLE 3.3: 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.

3.3 Types de fiches

3.4 Types de relations

3.5 Schémas

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" />
```

Chapitre 5

Syntaxe des fiches

Une fois un modèle de données décrit (même si cette opération est en pratique rare, puisque le modèle de données standard devrait généralement suffire), la construction d'un répertoire de fiches peut commencer.

La syntaxe est dérivée de Markdown et d'autres langages de formatage rapide. Il y a néanmoins quelques différences dans le comportement standard du parser :

— Tkacz utilise les *_tirets de soulignement_* pour la mise en italiques, et ***une seule étoile**** pour le gras.

— Un paragraphe indenté de quatre espaces ou plus n'est pas traité comme un bloc de code, mais est simplement indenté d'un niveau.

— Les blocs de code utilisent exclusivement la syntaxe « grillagée », en encadrant le bloc de ~~~ et en indiquant éventuellement le langage après la première série de tildes.

— La syntaxe des liens est supprimée. Les URLs sont automatiquement converties en liens.

— La syntaxe des blocs de description est modifiée.

— Le format des citations permet d'en préciser l'origine.

— Tkacz pourrait gèrer plusieurs tables des matières dans un seul document.

De plus, un certain nombre d'extensions spécifiques sont ajoutées.

5.1 Association clé-valeur

Toute fiche commence généralement par un préambule qui expose formellement son contenu. Ce préambule prend la forme

```
:name
```

```
    :first Michel
    :middle Paul :unused
    :last Foucault

:name Fuchs :aka
    Dans [Guibert 1980]

:birth 15 octobre 1926 @ Poitiers
:death 25 juin 1984 @ Paris
```

5.2 Liens et relations

```
<> friend [Didier.Éribon]
<> founder GIP
```

No newline at end of file

Chapitre 6

Sélection et recherche

La sélection et la recherche utilisent le mécanisme de la taxinomie pour rechercher des notes. Chaque taxon peut être conçu comme un ensemble de fiches. Les expressions de recherche prennent la forme suivante :

```
[Publications] 'Michel Foucault' date < {3 jan 1950}
```

La recherche répond à une logique globalement ensembliste ; les opérations fondamentales de la théorie des ensembles (intersection, union, différence, différence symétrique) forment les opérateurs principaux du mécanisme de recherche.

À terme, il est prévu de faciliter l'usage de ce mécanisme de recherche par une interface graphique d'élaboration des requêtes et/ou une formulation des requêtes dans un langage formalisé proche du langage naturel.

6.1 Syntaxe

La totalité des opérateurs peuvent manipuler quatre types de propriétés, soit les trois types d'ensembles :

`set` Un ensemble de fiches ou de taxons (c'est pareil, un taxon n'est qu'un ensemble de fiches)

`strset` Un ensemble de chaînes.

`attrset` Un ensemble de noms d'attributs ou de relations.

Et un type complexe, repéré par les accolades, spécifiques à certain type d'attributs, par exemple les dates.

Il n'existe pas de type « fiche unique » ou « chaîne » : tout est un ensemble, qui peut ne contenir qu'un élément.

6.2 Opérateurs de groupement

`[ ]` Sélectionne un taxon par son nom : `[str]`.

Les ambiguités peuvent être résolues en donnant un parent du taxon au format `[parent/taxon]`, un parent plus lointain `[parent/.../taxon]` ou le nom de la taxinomie `[@taxinomie: parent/taxon]` ou une combinaison : `[@taxinomie: parent//taxon]`.

" " Sélectionne une fiche par son nom.

Une fiche peut être aussi sélectionnée directement par son numéro.

Ces deux premiers opérateurs utilisent la virgule comme séparateur. `[taxon1, taxon2]` est un ensemble de taxons, et donc renvoie une valeur de type `set`.

() Les parenthèses augmentent la priorité d'une expression (cf. 6.6 page suivante). Rien de très original. L'expression `A+B*C` sera évaluée implicitement comme l'union de `A` et de l'intersection de `B` et `C`. Avec des parenthèses telles que `(A+B)*C`, elle renverra l'intersection de `C` et de l'union de `A` et `B`.

6.3 Opérateurs binaires

: `attribut:set` ou `attribut:str` retourne l'ensemble des fiches dont l'attribut `a` a au moins une valeur dans `expr`.

:: `a::expr` retourne l'ensemble des fiches dont l'attribut `a` a toutes ses valeurs dans expr.

< Strictement inférieur à, pour les attributs où cela à un sens.

<= Inférieur ou égal où, pour les attributs où cela à un sens.

> Strictement supérieur à, pour les attributs où cela à un sens.

>= Supérieur ou égal où, pour les attributs où cela à un sens.

& intersection ($\cap$). c'est l'opérateur implicite. `A&B` ou `AB` retournent l'intersection des taxons `A` et `B`. L'intersection est symétrique ; `A&B = B&A`

| union ($\cup$). `A|B` retourne l'ensemble des fiches de `A` et de `B`. L'union est symétrique ; `A|B = B|A`

- différence ($\setminus$). `A-B` renvoie l'ensemble `A` moins l'ensemble `B`. La différence n'est pas symétrique (l'intersection de `A-B` et `B-A` est vide : $(A \setminus B) \cap (B \setminus A) = \emptyset$.)

^ différence symétrique ($\triangle$). `A^B` renvoie la totalité des fiches de `A` ou `B` mais pas les deux. `A^B = (A+B)-(B*A)`.

6.4 Opérateurs unaires

- Inverse. `-expr` renvoie la totalité des fiches non contenues dans `expr`. `-expr = {*-expr}`

: équivalent approximatif de `:` sans spécifier le nom de l'attribut : `*:expr` renvoie toutes les fiches liées à expr. `* :expr` doit être un ensemble.

:* `expr`

6.5 Autres termes

`*` L'ensemble des fiches du dépôt.

`[ ]` Délimite une construction complexe spécifique à un type de données, par exemple une date.

`" "` Délimite une chaîne pour la recherche en plein texte. Isolé, il renvoie l'ensemble des fiches dans lesquelles ce texte a été trouvé ; sinon il peut être utilisé pour la recherche par attributs (`:, ::`)

`\` Caractère d'échappement.

6.6 Priorité

Les expressions sont évaluées avec les priorités suivantes. (1) et (2) précisent qu'il s'agit, respectivement, de la version unaire ou binaire d'un opérateur.

TABLE 6.1: Priorité des opérateurs de recherche

Priorité	Opérateurs
1	Opérateurs de groupement
2	Opérateurs unaires
3	&
4	— (binaire)
5	/

6.7 Synonymie

Pour des raisons de clarté, les opérateurs natifs ont les synonymes suivants :

TABLE 6.2: Synonymes des opérateurs de recherche

Opérateurs	Synonymes
`:`	`in`
`::`	
`&`	`and`
`|`	`or`

Opérateurs	Synonymes
`-`	`andnot`
`\`	`xor`
`*`	`all`

Les versions localisées pourraient implémenter ces synonymes dans leur langue.

Chapitre 7

Interface en ligne de commande

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
```

7.1 Arbre des commandes

```
tkacz
├── card ..... Opérations sur les fiches
│   ├── edit
│   ├── find
│   ├── new
│   ├── rm
│   └── view
├── coll ..... Opérations sur les collections
│   ├── edit
│   ├── new
│   ├── rm
│   └── view
├── repo ..... Opérations sur les dépôts
│   ├── new
│   └── edit
├── schema ..... Opérations sur le schéma
│   ├── add
│   └── extend
├── serve
└── shell
```

Deuxième partie

Implémentation

Chapitre 8

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.

8.1 Format d'un dépôt

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 :

```
/
├── .git
│   └── . . . ..... Dossier git, non géré par Tkacz
└── .tkacz
    ├── cache ..... Fichiers de cache
    │   └── db ..... Fichiers SQLite
    ├── schemas ..... Code Python des schémas .2 . . .
    └── manifest ..... Déclarations Tkacz de base (JSON)
```

Le fichier `.tkacz‣manifest` est une représentation JSON de la structure détaillée ci-dessous. évidemment

```
{
    "tkacz": {
        "frameworkVersion": [0,1,0],
        "formatVersion":    [0,1,0],
        "schemaId":         "core",
        "schemaVersion":    [0,1,0]
```

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

```
    },
    "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).

8.2 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].

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.

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]
    }
}
```

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.*

8.3 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 9

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

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.

Liste des tableaux

Table des matières