PyUseOCL Documentation

Release 0.8.1

megaplanet

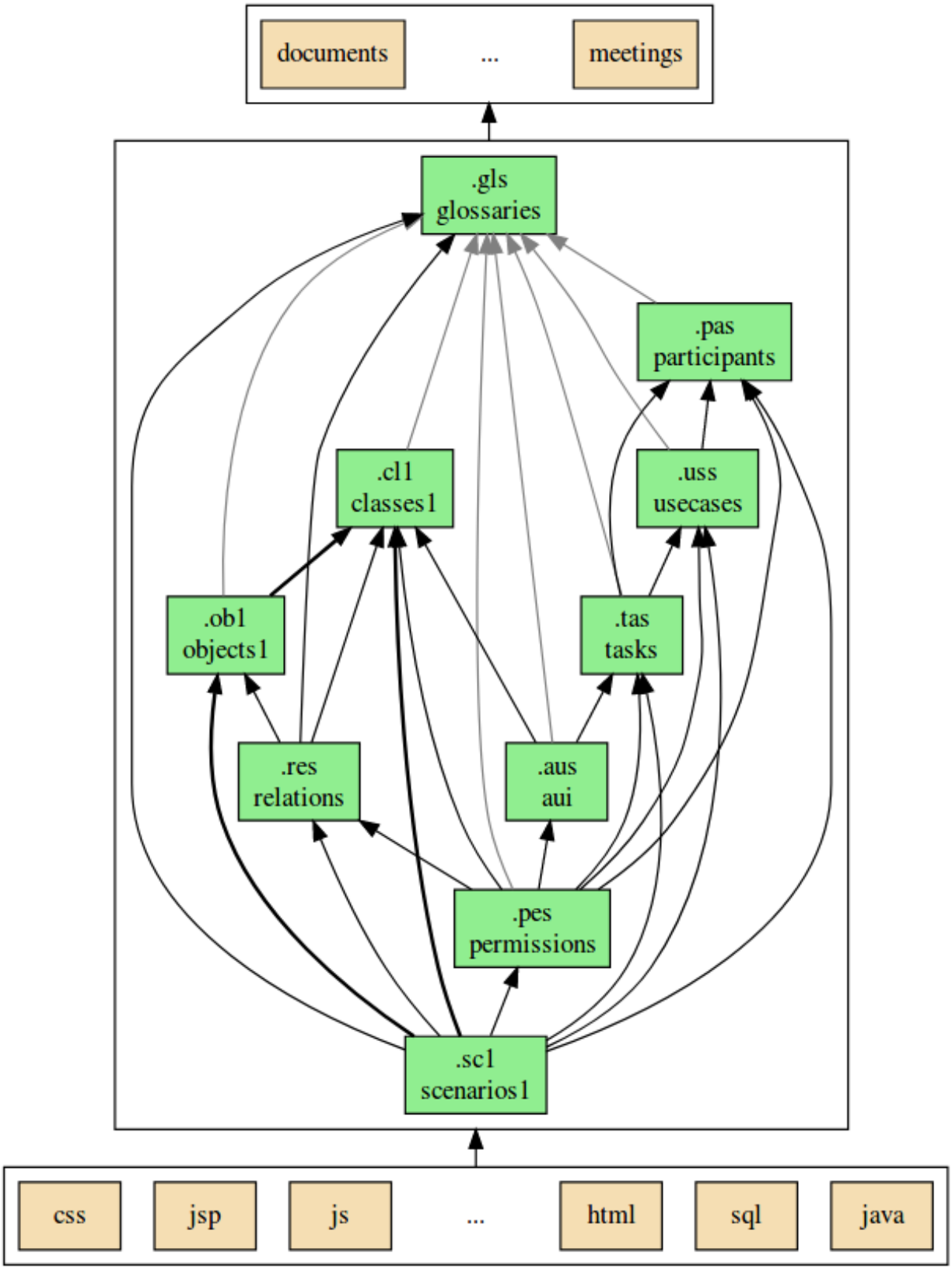
Mar 08, 2021

Contents

ModelScript is a lightweight modeling environment based on:

- a dozen of small textual **languages** called model scripts,
- an underlying modeling **method**,
- a few set of **tools**.

The image below shows the dependency graph between the different ModelScript languages:


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

CHAPTER 1

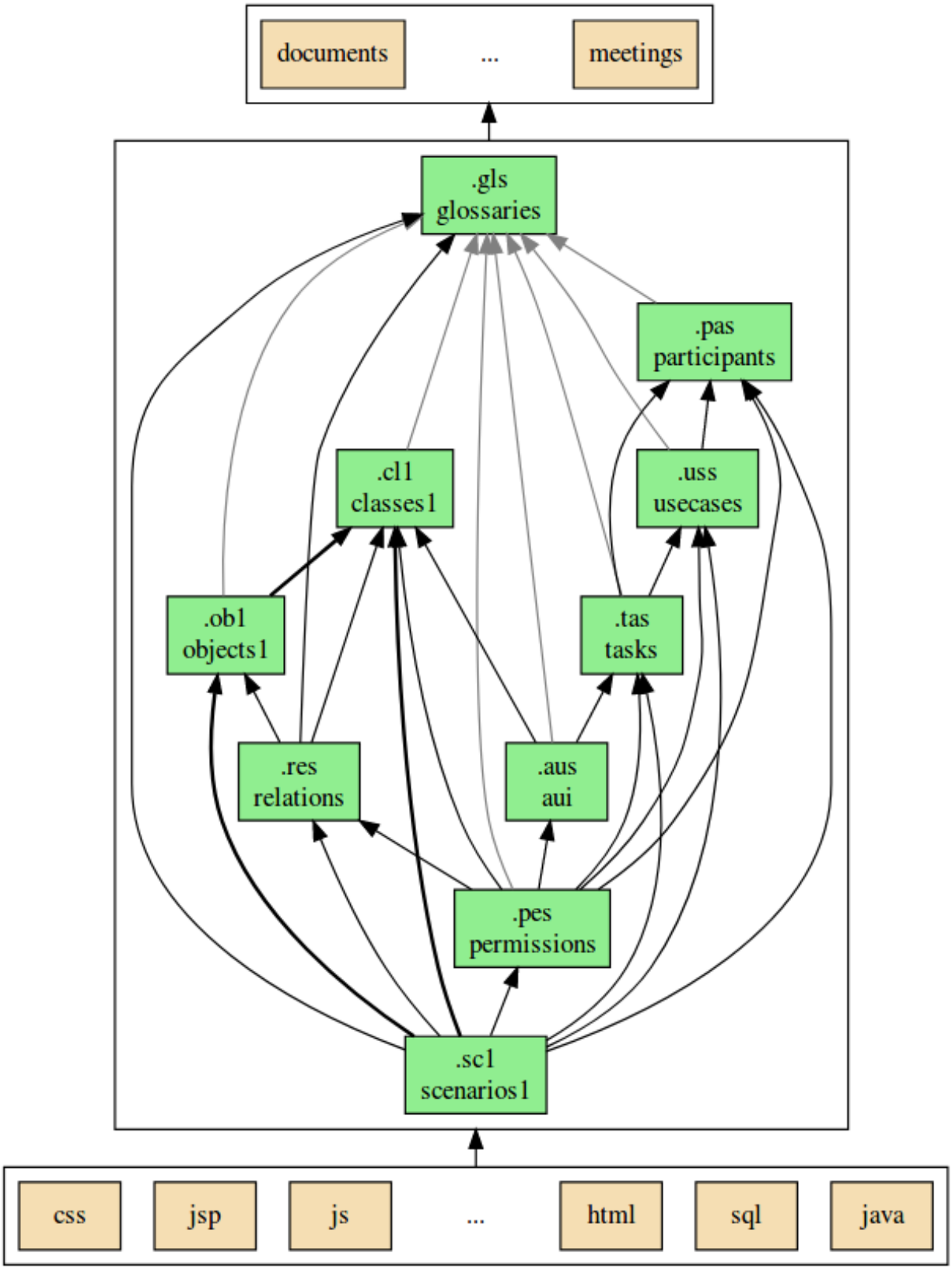
Overview

1.1 ModelScript

ModelScript is a lightweight modeling environment based on:

- a dozen of small textual languages called model scripts
- an underlying (yet truely optional) modeling methodology.

The image below shows the dependency graph between the different ModelScript languages:


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

1.1.1 Objectives

ModelScript is intented to be used in classrooms.

ModelScript is currently developed to support courses at the University of Grenoble Alpes. Course topics include:

- software engineering,
- database design,
- user interface design,
- model driven engineering,
- information systems.

1.1.2 ModelScript1, first prototype

The current version of ModelScript, referred as ModelScript1, is a very first prototype. **ModelScript1 is limited to syntaX checking *apart for three languages** that are based on the USE OCL tool.

The next version, under development, will:

- implement the full language semantics of all languages,
- replace USE OCL languages by custom ones,
- add additional features such as automated document generation.

The languages *ClassScript*, *ObjectScript* and *ScenarioScript* are actually implemented using USE OCL. These languages are named *ClassScript1*, *ObjectScript1*, *ScenarioScript1*. Each of these languages are based on a few annotations embedded on USE OCL comments.

Note also that TaskScript is just a convenient alias here for the language underlying the Kmade environment from university of Nancy. TaskScript is the only language with no textual syntax.

1.1.3 ModelScript ecosystem

ModelScript languages are listed below. All languages exist in the form of a textual syntax (some of them having a graphical syntax as well), apart for tasks diagrams that have only a graphical syntax.

language	main concepts
GlossaryScript	*entries, packages, synonyms, translations...*
ClassScript1[1]	*classes, attributes, inheritance, associations, invariants...*
ObjectScript1[1]	*objects, slots, links, link objects*
RelationScript	*relations, columns, keys, constraints, dependencies...*
ParticipantScript	*actors, personas, persons, roles, stakeholder*
UsecaseScript	*actors, usecases, interactions*
TaskScript[2]	*tasks, task decomposition, task decorations...*
AUIScript	*spaces, links, concept references*
PermissionScript	*subjects, resources, permissions, actions*
ScenarioScript1	*scenarios, contexts, usecase instances, persona, step...*

[1] *ClassScript1*, *ObjectScript1* and *ScenarioScript1* are currently annotated versions of the USE OCL language.
[2] The Kmade modeling environment is used for task models. There is no textual syntax. "TaskScript" is just used here for the sake of consistency.

1.1.4 Language graph

TODO

CHAPTER 2

Languages

2.1 Languages

2.1.1 GlossaryScript

Exemples

L'exemple ci-dessous n'a aucun sens. Il est seulement utilisé pour illustrer la syntaxe du langage GlossaryScript.

Note: Le glossaire fourni ci-dessous n'est pas complet. Certaines définitions sont manquantes bien qu'elles soient référencées.

```
glossary model Medium
    | `Description` de `un` élément
    | dans `un` contexte `uno` et `deux`
    | `un` `test`


//-------------------------------------------------------------
//   Glossaire du domaine
//-------------------------------------------------------------

package GlossaireDuDomaine

FilDeDiscussion
    | Suite ordonnée de `Messages`
    synonyms: Uno One
    inflections: unite uns
    texts:
        fr: "Fil de discussion"
        en: "Thread"
```

(continues on next page)

(continued from previous page)

```
Reference
    | Mot ou suite de mots faisant référence à
    | un `Concept` déjà défini. Attention à l'`Indentation`
    | qui doit être toujours de `huit` espaces.
    synonyms : a b c

//-------------------------------------------------------------
//   Glossaire technique
//-------------------------------------------------------------

package GlossaireTechnique

MVC
    | Patron de conception utilisé lors de la définition
    | d'interface homme machine.
    | Voir https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
    | In the context of this project ...
```

GlossaryScript

Le langage `GlossaryScript`, tel que défini ci-dessus par l'exemple, permet d'exprimer des *glossaires*. Plusieurs glossaires peuvent être défini dans un même modèle. Par exemple il est possible de définir un glossaire de domaine bancaire et à coté de cela un glossaire technique définissant les termes de J2EE. Tous les glossaires sont définis dans le même fichier. La notion de paquetage (packages) permet de séparer ce fichier en plusieurs glossaires. Les scripts GlossaryScript ont l'extension `.gls`.

Concepts

Un **glossaire** est une collection d'**entrées** optionnellement organisée en **paquetages**. Le but d'un glossaire est de définir tous les **termes** utilisés dans le contexte d'un projet donné.

Un **glossaire** est composé de :

- un ensemble d'**entrées** composé d'un **terme principal** ainsi que de **termes alternatifs** (**synonymes**, **abbréviations**, etc.),
- la définition des relations entre ces différents termes,
- une définition pour chaque entrée, définition pouvant faire référence aux différents termes du glossaire.

Entrées

Une **entrée** est essentiellement :

- un **terme principal** (p.e. `Fil` dans l'exemple ci-dessous)
- un ensemble de **termes alternatifs** (**synonymes**, **abbréviations**, etc.)
- une **définition** qui correspond bien à l'ensemble des termes,
- un ensemble optionnel de **traductions** définissant pour différents **langages** la représentation concrète de l'entrée.

```
Fil
    | Séquence de `Messages` en réponse à un `Initial`. Un fil
    | peut être `Bloque` ou `Ouvert` et est identifié par
    | un `Theme` et un ensemble de `Cles`.
    synonyms: Discussion, FilDeDiscussion
    inflections; Fils
    translations
        fr: "fil de discussion"
        en: "thread"
        es: "conversacion"
...
```

Le **terme principal** (`Fil` ici) est celui sensé être référencé :

- dans le reste du glossaire,
- et dans les textes techniques lorsque ceux-ci sont réécrits avec le glossaire.

Note: Un même mot peut parfois avoir plusieurs acceptions (plusieurs significations). Si c'est le cas numéroter les termes principaux pour chaque acception. Par exemple `Fil1` et `Fil2` peuvent correspondre à deux acceptions du mot Fil. Les différentes occurrences du mot Fil dans les différents textes devront bien évidemment être remplacées par `Fil1` ou `Fil2` .

Synonymes

Plusieurs synonymes peuvent être associés à une entrée :

```
Fil
    | Définition
    | ...
    synonyms: Discussion, FilDeDiscussion
```

Les **synonymes** sont des termes qui ont la même signification que le terme principal. Par exemple dans l'exemple ci-dessus `Discussion` et `Fil` ont la même signification, mais `Fil` est le terme principal. Cela signifie que toutes les occurrences de `Discussion` dans les textes devraient être remplacées par `Fil`.

Inflexions

Les **inflexions** sont des dérivations du terme principal, tel que pluriels, formes avec des genres différents, formes verbales vs. nominales, conjugaisons, etc.

```
Fil
    | Définition
    | ...
    inflections: Fils
```

Au contraire des synonymes les inflexions sont des variations "normales" du terme principal et ne sont pas supposées être remplacé par celui-ci.

Traductions

Alors qu'une entrée est définie par son terme principal, cette entrée peut posséder plusieurs **traductions**. Chaque traduction est définie par :

- la langue utilisée pour la traduction (encodée en iso-639),
- la chaîne de caractères correspondant à la traduction.

```
Fil
    translations
        fr: "fil de discussion"
        en: "thread"
        es: "conversacion"
```

Paquetages

Un ensemble d'entrées peut être scindé en plusieurs **paquetages** en utilisant le mot clé `package` suivi du nom de paquetage. Deux possibilités sont offertes par ModelScript : (1) utiliser le mot clé de manière globale suivi d'un ensemble d'entrées, ou (2) d'indiquer pour chaque entrée le paquetage à laquelle elle appartient.

Paquetages globaux

Toutes les entrées entre un mot clé `package` et le suivant sont rangées dans ce paquetage. De plus toutes les entrées avant le premier mot clé `package` font partie du paquetage `unamed`.

Note: Pour éviter une indentation supplémentaire les entrées et les paquetages sont définis au même niveau.

```
glossary model CyberForum


//-------------------------------------------------------------
//   Glossaire du domaine
//-------------------------------------------------------------

package GlossaireDuDomaine

Forum
    ...
Fil
    ...
...

//-------------------------------------------------------------
//   Glossaire technique
//-------------------------------------------------------------

package GlossaireTechnique

MVC
    | Patron de conception utilisé lors de la définition
    | d'interface homme machine.

...
```

Paquetages en ligne

Une entrée peut être à n'importe quel moment associée à un paquetage particulier, existant ou non. Il suffit d'utiliser pour cela le mot clé `package` à l'intérieur de l'entrée ; voir par exemple `Numbers` ci-dessous :

```
...
package Letters                 // Toplevel package

Alpha

One
    package: Numbers            // Inline package

Beta
```

Règles

Les règles suivantes doivent être appliquées dans l'élaboration des glossaires :

- Dans les définitions, les références à d'autres termes du glossaire doivent être entre backquotes (p.e. `` `Backquote` ``). Ces termes doivent être définis.
- Dans les textes les mots sans `` `Backquote` `` font référence aux mots du dictionnaire. Les deux peuvent cohéxister.
- Les définitions doivent normallement commencer par une forme nominale, tout comme dans un dictionnaire. La définition `"Singe : Animal ...` est adaptée. Le premier terme ("Animal" ici) peut faire partie du glossaire entre backquotes ou être un terme d'usage courant (sans backquotes).
- Toutes les définitions doivent correspondre au contexte particulier du projet. Omettre les définitions générales. Par exemple "Personne : Etre humain" n'apporte rien si le terme "Personne" n'a pas de signification différente de "personne" d'usage courant. Mettre "Personne" dans le glossaire s'il s'agit d'un terme spécifique au projet.

Réécriture de textes

Au fur et à mesure qu'un glossaire est défini, il faut réécrire les textes utilisant "informellement" le glossaire. En pratique pour chaque terme apparaissant dans un texte il faut déterminer s'il s'agit :

- d'un terme d'usage général : aucune action n'est nécessaire.
- d'un terme du domaine mais non défini : l'ajouter au glossaire.
- d'un terme déjà défini comme terme principal dans le glossaire. il faut alors créer une référence (entre backquotes) vers ce terme.
- d'un synonyme déjà défini : il faut le remplacer par le terme principal entre .

Ce travail de réécriture / définition du glossaire est bien évidemment itératif. L'objectif final est d'obtenir des textes les moins ambigüs et plus cohérents possible avec le glossaire.

Réécriture des identificateurs

La plupart des identificateurs (UML, Class, Java, SQL, etc.) devraient faire référence à un ou plusieurs termes d'un glossaire du domaine et/ou technique. C'est le cas par exemple pour l'identificateur suivant :

getCartLayout

Le terme `Cart` provient sans doute du glossaire du domaine alors que `Layout` peut provenir d'un domaine technique correpondant à un framework utilisé.

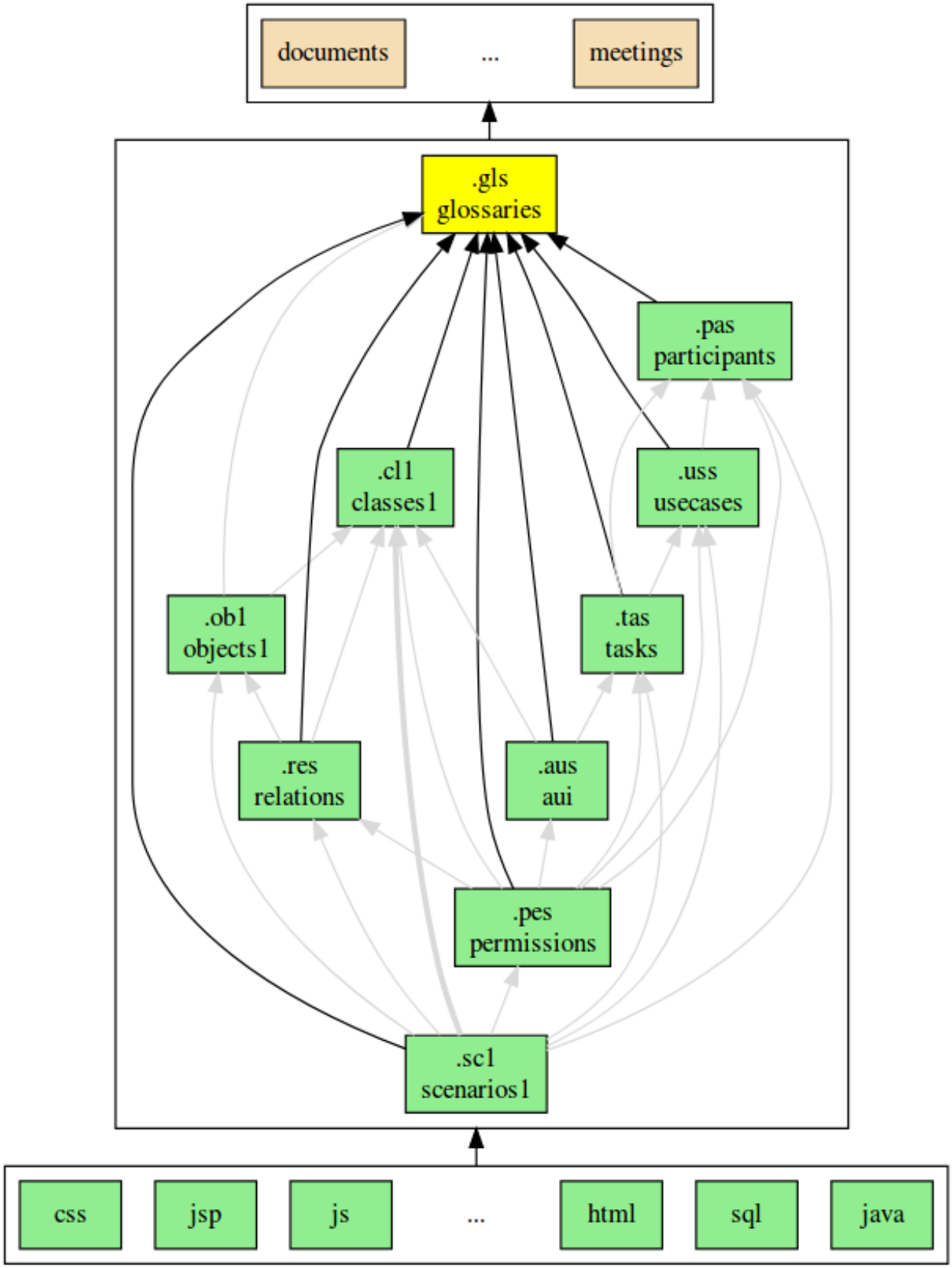
Dans certains cas des abbréviations sont utilisées pour obtenir des identificateurs plus courts. Celles-ci doivent être ajoutées dans le glossaire technique (p.e. `DAO`) ou dans le glossaire de domaine (`num` pour `numéro`). Le glossaire doit assurer l'usage des termes de manière homogéne et consistante dans tous les modèles et dans tout le code.

Un identificateur qui ne fait référence ni au domaine ni aux aspects techniques est sujet à suspicion.

Dans tous les cas il est fondamental, lorsque les glossaires changent ou lorsque de nouveaux identificateurs sont définis, de s'assurer de l'alignement entre glossaire et les autres artefacts.

Dépendances

Le graphe ci-dessous montrent les dépendances entre langages et en particulier avec le glossaire. Comme on peut le voir le glossaire dépend de tous les éléments issus de la capture des besoins. Le glossaire est en fait extrait des différents documents existants. Dans la direction opposée tous les modèles dépendent du glossaire dans la mesure où tous ces modèles peuvent avoir de la documentation et sont certainement basés sur des identificateurs.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

2.1.2 TrackScript

Exemples

```
track model CyberBibliobus
import glossary model from '../glossaries/glossaries.gls'

question Q1: Catalogue des oeuvres
    | Il est fait mention des `Oeuvres` et de leurs `Auteurs` mais
    | le moyen de créer/maintenir ces informations n'est pas précisé.
    | Qui crée/maintient le `Catalogue` ?
    | Est-ce un système externe avec lequel `CyberBibliotheque` doit
    | s'interfacer ?
    github: #3
    priority: low
    status: closed
    who: ERT PGI NZW
    conclusion
        | La gestion du `Cataloque` des `Oeuvres` est en dehors
        | du périmètre de la version v3.2. Pour l'instant le
        | `Catalogue` sera figé et fourni via un fichier XML.

question Q2: Rendu en retard
    | Aucune information n'est fournie sur le mode de `Rendu`
    | dans le cas ou celui-ci se fait en retard. Le contenu
    | de la ligne [A4] doit être précisé avant de pouvoir réaliser
    | la définition du cas d'utilisation `RendreUnItem`.
    status: open
    who: KET MER

hypothesis H1: Transfert de stocks
    | On suppose que le transfert de `Stocks` se fait "en dehors"
    | de `CyberBibliotheque` [A31][A32] et que la seule fonctionnalité
    | qui doit être développée est le fait que les `Magaziniers`
    | incrémentent/décrémentent les `Stocks` de leur `Bibliotheque`
    | respective [A33].
    github: #12
    status: validated
    date: 2020-05-21
    who: NZW

hypothesis H2: Emprunt enseignant de 30 jours
    | Les lignes [A23] et [A26] semblent contradictoires. Il semble
    | logique de ne pas restreindre le `Personel` à la contrainte
    | des 15 jours indiquée en [A23].
    priority: low
    status: open

decision D1: Pas de transfert des livres
    | La gestion du `Transfert` des `Livres` ne sera
    | pas prise en compte avant la version v2.
    date: 2020-05-21
    who: ADZ NZW PGI ZSE

decision D2: Entree en retard de plus d'un jour
    | La `Rentree` d'une `Oeuvre` en `Retard` de plus d'un
    | jour doit être prise en compte contrairement à ce
```

(continues on next page)

(continued from previous page)

```
    | que peut laisser supposer la ligne [A34].
    date: 2020-05-21
    who: ADZ NZW PGI ZSE

impediment I1: Pas de diagrammes via USE OCL
    | Le logiciel USE OCL fonctionne en mode textuel
    | mais pas en mode graphique. Il est donc impossible
    | à ce stade de créer des diagrammes de classes et des
    | diagrammes d'objets. Les tâches concepts.classes.diag
    | et concepts.objets.diag sont en attente.
    action: A1
    date: 2020-03-18
    who: ADZ JFE

impediment I2: Pas de salle de réunion disponible
    | La salle de réunion allouée à l'équipe n'est généralement
    | pas disponible le vendredi pour les retrospectives.
    date: 2020-03-20
    who: NZW

action A1: Contacter le service informatique pour USE OCL
    impediment: I1
    who: JFE

action A2: Restructurer le fichier classes.cl1 et ob1.ob1
    | Les noms des classes doivent être revus et alignés
    | au glossaire
    github: #15
```

TaskScript

Le modèle de suivi peut être utilisé dans de multiples contextes :

- **ordres du jour** de réunions. Le client n'étant pas disponible en permanence, les questions et hypothèses doivent être consignées et sérialisées. Ces différents points peuvent ensuite être soulevés lors d'une prochaine réunion avec le "client". Un tel modèle peut donc être utilisé pour établir l'ordre du jour d'une réunion future.
- **compte rendus**. Il est possible de définir des "décisions" et des "actions" dans le modèle de suivi. Bon nombre de décisions sont prises lors de réunions, et ces décisions peuvent être référencées dans les comptes rendus de réunions. Les actions à entreprendre font aussi partie des conclusions des réunions.
- **traçabilité**. Le modèle de suivi sert de support à la traçabilité tout au long du projet. Il est par exemple possible de déterminer quelles personnes, quelles parties prenantes sont ou ont été impliquées dans telle ou telle décision.

Note: Notons que TaskScript recouvre partiellement ce qui peut être exprimé habituellement via des issues GitHub. Autrement dit certains éléments de suivis (questions, hypothèses, actions, etc.) peuvent être matérialisés sous forme d'issues. Il s'agit alors de définir quel est la source d'information principale, GitHub ou le modèle `suivis.trs`, et pour quelle catégorie de suivis. Dans certain cas il est pertinent de faire référence aux issues GitHub à partir du modèle de suivis (via une référence comme `#13`).

Concepts

Le modèle de suivi a pour objectifs de consigner différents *points de suivis* :

- des **questions**,
- des **hypothèses**,
- des **décisions**,
- des **empêchements**,
- des **actions**,
- des **problèmes**.

La différence entre ces différents points de suivi sont définis ci-dessous.

Questions

Les **questions** sont des interrogations que les membres de l"équipe peuvent avoir à propos d'une partie du projet. Par contraste avec les *hypothèses*, une *question* a un certain caractère bloquant : aucune supposition n'est faite ; la question doit être répondue.

Hypothèses

En cas de doute les membres de l'équipe peuvent émettre des **hypothèses** lorsqu'un point du projet n'est pas clair. Ces *hypothèses* permettent à l'équipe de continuer à travailler. Chaque *hypothèse* est enregistrée de manière à être validée ou invalidée lors d'une réunion avec le client par exemple. Lorsqu'une *hypothèse* est émise l'équipe prend un risque par rapport à tous les développements basés sur cette *hypothèse*. Evaluer ce risque est fondamental. Si trop de développements dépendent d'une *hypothèse* il est sans doute préférable de poser une *question* et d'attendre la réponse.

Décisions

Dans un projet, différentes **décisions** sont prises à différents moments du cycle de vie. Ce peut être le cas lors de réunions entre différentes parties prenantes. Il est essentiel de rendre explicite le contenu de la décision, la date à laquelle elle a été prise, qui a pris cette décision, qui l'a validé, etc. Un compte rendu de réunion fait typiquement référence à une série de décisions. D'autres décisions peuvent être prises à d'autres moments par le client ou l'équipe de développement.

Empêchements

Le déroulement d'un projet est parfois freiné par des **empêchements**. Un *empêchement* correspond à un problème qui survient dans le déroulement d'un projet et qui limite ou empêche certaines tâches de progresser normallement. Ce peut être l'indisponibilité d'une salle de réunion, l'indisponibilité d'un serveur, le fait qu'une question n'a pas été répondue et que cela devienne un caractère bloquant, etc. Un *empêchement* signale à un interlocuteur (tel qu'un chef de projet par exemple) qu'une action doit être menée pour contrecarrer cet *empêchment*. Identifier et lister les *empêchements* est un élément important de la méthode Scrum. Les *empêchements* sont typiquement identifiés au cours des *standup meetings*

Actions

Les actions correspondent aux actions devant être réalisées et étanr typiquement consignées suite à une réunion, comme parexemple un *standup meeting*, une *retrospective* ou une *audit*.

Problèmes

Le développement de tout projet soulève, à un moment ou à un autre, différents **problèmes**. Ces *problèmes* doivent être identifiés, décrits, traités, suivis, etc. Le terme "problème" est volontairement générique. Tombent dans cette catégorie tous les éléments de suivis n'étant pas dans une autre catégorie plus spécifique.

Règles

- Chaque point de suivi doit être identifié de manière unique. Par exemple `D3`, `Q3`, `H12`, `I2`, `P2`, etc.
- Réferencer ces identificateurs entre crochets (e.g. `[H12]`) dans le(s) modèle(s) impactés. En commentaire ou via tout autre moyen adapté au langage utilisé.
- La formulation des points de suivis doit impérativement être précise et faire référence aux termes définis dans le glossaire (entre backquotes "`"). C'est le cas notamment des questions et hypothèses qui sont à destination du client.
- Les points de suivis doivent avoir un titre court mais le plus explicatif possible.
- Les points de suivis doivent être aussi pertinents que possible du point de vue des différentes parties prenantes impliquées. Par exemple ne pas utiliser de vocabulaire technique si un point de suivi est adressé à un client.
- Les points de suivis doivent se référencer entre eux lorsque nécessaire (par exemple une action fera peut être "suite" à une décision). Les références vers les issues GitHub peuvent aussi être utiles.

2.1.3 ClassScript1

Exemples

Note: L'exemple suivant n'a strictement aucun sens. Il est juste fourni ici pour donner une idée de la syntaxe de ClassScript1.

```
--@ class model Jungle
--@ import glossary model from "../glossaries/glossaries.gls"

model Jungle

enum Season {
    spring,
    summer
}

class Yellow
end

class Banana < Yellow
    --| A Banana is a nice Fruit that growths
    --| in the forest.
    attributes
        _name_ : String --@ {id} {derived} {optional}
        length : Integer
            --| the length of the banana expressed in milimeters.
        size : Real
        frozen : Boolean
```

(continues on next page)

(continued from previous page)

```
        expirationDate: String --@ {Date}
        growthTime : Season
end

association Owns
    --| A person owns some cars if he or she
    --| bought it and didn't sell it.
    between
        Person [1] role owner
        Car[*] role properties
            --| A person can have several
            --| properties if he or she's lucky
end

associationclass Hate
    between
        Monkey [*] role monkeys
        Snake [*] role snakes
    attributes
        reason : String
        intensity : Integer
end

constraints

--@ constraint SmallBananas
--@     scope
--@         Banana.size
--@         Banana.length
--@     | Bananas are longer than their length.

        context self : Banana inv SmallBananas :
        self.size > self.length

--@ constraint MomentConcerne
--@     scope
--@         Atelier.dateDeDebut
--@         Atelier.dateDeFin
--@         Concerne
--@         Emprunt.dateDeSortie
--@     | Si un emprunt <e> concerne un atelier <a> alors cet
--@     | emprunt <e> a eu lieu dans la période <p> correspondant
--@     | à l'atelier <a>.
```

ClassScript1

ClassScript est un langage textuel pour les diagrammes de classes UML. Dans la version de ModelScript le langage `ClassScript1` est en fait une version augmentée d'un sous ensemble du langage USE OCL. ClassScript1 diffère très légèrement de USE OCL:

- annotations. Deux types d'annotations sont ajoutées sous forme de commentaire USE OCL :
  - `--|` préfixe la documentation ModelScript.
  - `--@` préfixe les autres annotations ModelScript.

- restrictions: ClassScript1 ne prend pas en compte les associations qualifiées et les autres fonctionnalités telles que les post-conditions et les post-conditions.

Alors que l'extension `.use` est utilisée dans le cadre de USE OCL, ici `.cl1` est l'extension des scripts ClassScripts1.

Outils

Analyse de modèles

Les modèles ClassScript1 peuvent être utilisés avec l'outil USE OCL. Quand la *méthode ModelScript* est utilisée la ligne de commande suivante permet de "compiler" le modèle de classes (en supposant que le répertoire courant est le répertoire racine du projet de modèlisation) :

```
use -c concepts/classes/classes.cl1
```

L'interpréteur vérifie si il y a des erreurs ou non. Ce peut être des erreurs de syntaxe, des erreurs de types, des contraintes violées, etc. Si aucune erreur n'est affichée alors le modèle de classes est correct.

Génération de diagrammes

Dessiner un diagramme de classes UML est possible avec l'outil USE OCL.

```
use -nr concepts/classes/classes.cl1
```

Voir la page "créer un diagramme de classes UML" pour plus d'information.

Quand la *méthode ModelScript* est utilisée le fichier de "layout" de USE OCL (la disposition des classes) doit être sauvegardé dans le fichier `concepts/classes/diagrammes/classes.cld.clt`. Un copie d'écran du diagramme doit être réalisée et il s'agit de remplacer le fichier `concepts/classes/diagrammes/classes.cld.png`.

Concepts

Un modèle de classes est basé sur les concepts suivants :

- énumérations,
- classes,
- attributs,
- associations,
- classes associatives,
- contraintes.

Enumérations

```
enum Season {
    --| Documentation of the enumeration
    --| Explains what is a season.
    winter,
        --| Documentation of the
```

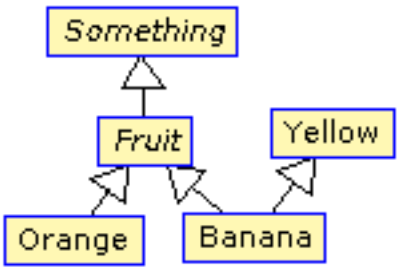
(continues on next page)

(continued from previous page)

```
        --| winter value
    autumn,
        --| Documentation of the autumn value
    spring,
    summer
}
```

Classes

Diagramme de classes UML:



ClassScript1 (basé sur USE OCL):

```
class Yellow
    --| Documentation of the
    --| yellow class
end

abstract class Something
    --| Something is an abstract class
end

abstract class Fruit < Something
    --| Fruits are particular cases of Something
end

class Banana < Fruit, Yellow
    --| Bananas are both fruits and
    --| yellow things.
end
```

Attributs

ClassScript1 (basé sur USE OCL):

```
class Banana
    --| A Banana is a nice Fruit that growths
    --| in the forest.
    attributes
        _name_ : String --@ {id} {derived} {optional}
            --| A banana always have nice names.
        length : Integer
            --| The length of the banana
```

(continues on next page)

(continued from previous page)

```
            --| is between 5 and 40
        size : Real
        frozen : Boolean
        expirationDate: String --@ {Date}
        growthTime : Season
        remainingDays : Integer
end
```

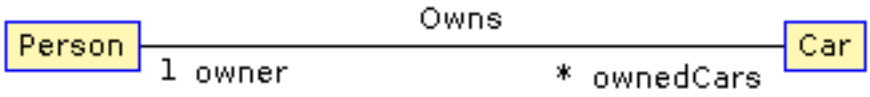
Attribute types Les attributs peuvent avoir les types suivants (lire la note sur les Dates pour plus de détails) :

- `Boolean`,
- `Integer`,
- `Real`,
- `String`,
- `Date`,
- `DateTme`,
- `Time`,
- une énumération.

Dates Les types `Date`, `DateTime` et `Time` n'existent pas en USE OCL. Les attributs de ces types doivent donc être défini comme étant de type `String` et les annotations `{Date}`, `{DateTime}` or `{Time}` doivent être ajoutées dans le code (voir l'exemple ci-dessus). La valeur de ces attributs doivent être représentés comme suit : `2020/12/23` pour les valeurs dy type Date, `2020/12/23-23:50:59` pour DateTime, and `23:00:32` pour Time. Ce format permet les comparaisons. Les autres opérations ne sont pas possibles.

Associations

UML class diagram:



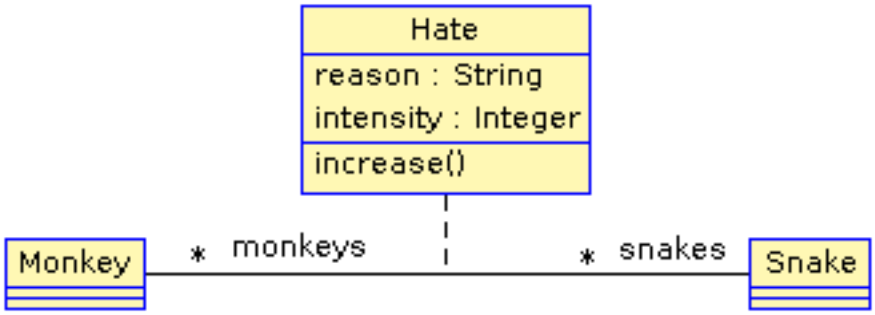
ClassScript1 (basé sur USE OCL):

```
association Owns
    --| A person owns some cars if he or she *
    --| bought it and didn't sell it.
    between
        Person [1] role owner
        Car[*] role properties
            --| A person can have several
            --| properties if he or she's lucky
end
```

Notons que l'ordre des roles est important. Dans l'exemple ci-dessus l'association se lit "(an) owner Owns (some) ownedCars" : le premier rôle est le sujet de la phrase ; le second rôle est le complément. L'ordre des rôles est également important pour la création des liens dans les diagrammes d'objets.

Association Classes

UML Diagram:



ClassScript1 (basé sur USE OCL):

```
associationclass Hate
    --| Some monkeys hate some snakes.
    --| That's life. Life in the jungle.
    between
        Monkey [*] role monkeys
        Snake [*] role snakes
    attributes
        reason : String
        intensity : Integer
end
```

Contraintes

USE OCL permet l'écriture de 3 types de contraintes : invariant, pré-conditions et post-conditions. Par contre ClassScript1 est basé sur l'utilisation d'invariants uniquement. Par abus de language on utilisera de manière interchangeable les termes "contrainte" et "invariant".

En ClassScript1 les contraintes (invariants) peuvent être définies en langage naturel en respectant toutefois un certain format. Ces contraintes peuvent ensuite être décrites en langage OCL.

Contraintes en Langage Naturel (LN)

Ecrire les contraintes en Langue Naturelle (LN) est une étape indispensable avant de formaliser ces contraintes en OCL. C'est en effet le client qui exprime ces contraintes, ou tout au moins qui les valide.

Structure

Chaque contrainte doit comporter les éléments suivants :

- un **identificateur** (p.e. `FormatMotDePasse`),
- une **portée** d'application (mot clé `scope`), c'est à dire la partie du diagramme de classes qui permet d'expliquer "où se trouve" la contrainte. La zone est représentée par une liste de noms de :
  - **classes** (p.e. `Personne`),
  - **associations** (p.e. `Concerne`),

  - **attributs** (p.e. `Personne.nom`),
  - **roles** (p.e. `Personne.parents`).

- une **description** en langue naturelle. Idéalement la description doit pouvoir être lue par le "client' aussi bien que par les développeurs. La description doit à la fois faire référence au glossaire, mais également autant que possible aux identificateurs se trouvant dans le diagramme. La correspondance entre les éléments décrivant la portée du modèle doit être claire et non ambigüe.

Exemple

Dans cet exemple la contrainte est un invariant. Ce code est à ajouter à la fin du modèle de classes (à la fin du fichier `classes.cl1`).

```
--@ invariant MomentConcerne
--@     scope
--@         Atelier.dateDeDebut
--@         Atelier.dateDeFin
--@         Concerne
--@         Emprunt.dateDeSortie
--@     | Si un emprunt concerne un atelier alors cet
--@     | emprunt a eu lieu dans la période correspondant à
--@     | l'atelier.
```

Dans l'exemple ci-dessus la notion de période n'est pas nécessairement claire et la locution "a eu lieu" non plus. Il est possible de préciser la phrase comme ci-dessous. Par ailleurs ci-dessous l'utilisation de variables a été ajoutée sous forme de "marqueurs". Il s'agit donc de langue naturelle "marquée". Ces variables ne sont pas nécessaires dans cet exemple mais elles peuvent être utiles avec des phrases plus complexes. Elles peuvent également se réveler utiles pour guider d'une part l'implementation de la contrainte et d'autre par l'écriture des tests positifs et négatifs.

```
--@     | Si un emprunt <e> concerne un atelier <a> alors
--@     | la date de sortie de l'emprunt <e> eu lieu entre la date de
--@     | début <dd> de l'atelier <a> et sa date de fin <df>.
```

Méthode

Trouver les contraintes à définir peut s'avérer difficile dans le cas de problèmes complexes. L'une des techniques possibles est de passer un à un les différents éléments d'un modèle de classes. Il s'agit ainsi de lister les contraintes portant sur :

- **un attribut**, typiquement les contraintes de domaine (e.g. *age>0*)
- **plusieurs attributs** d'une classe (e.g. `min<=max`)
- **une association** (e.g. *le père d'une personne <p> est plus agé*)
- **plusieurs associations** (e.g. *le salaire d'une personne employée dans une entreprise ne peut pas être supérieur à 5% du budget du projet sur lequel elle travaille, sauf si elle est classée A*).

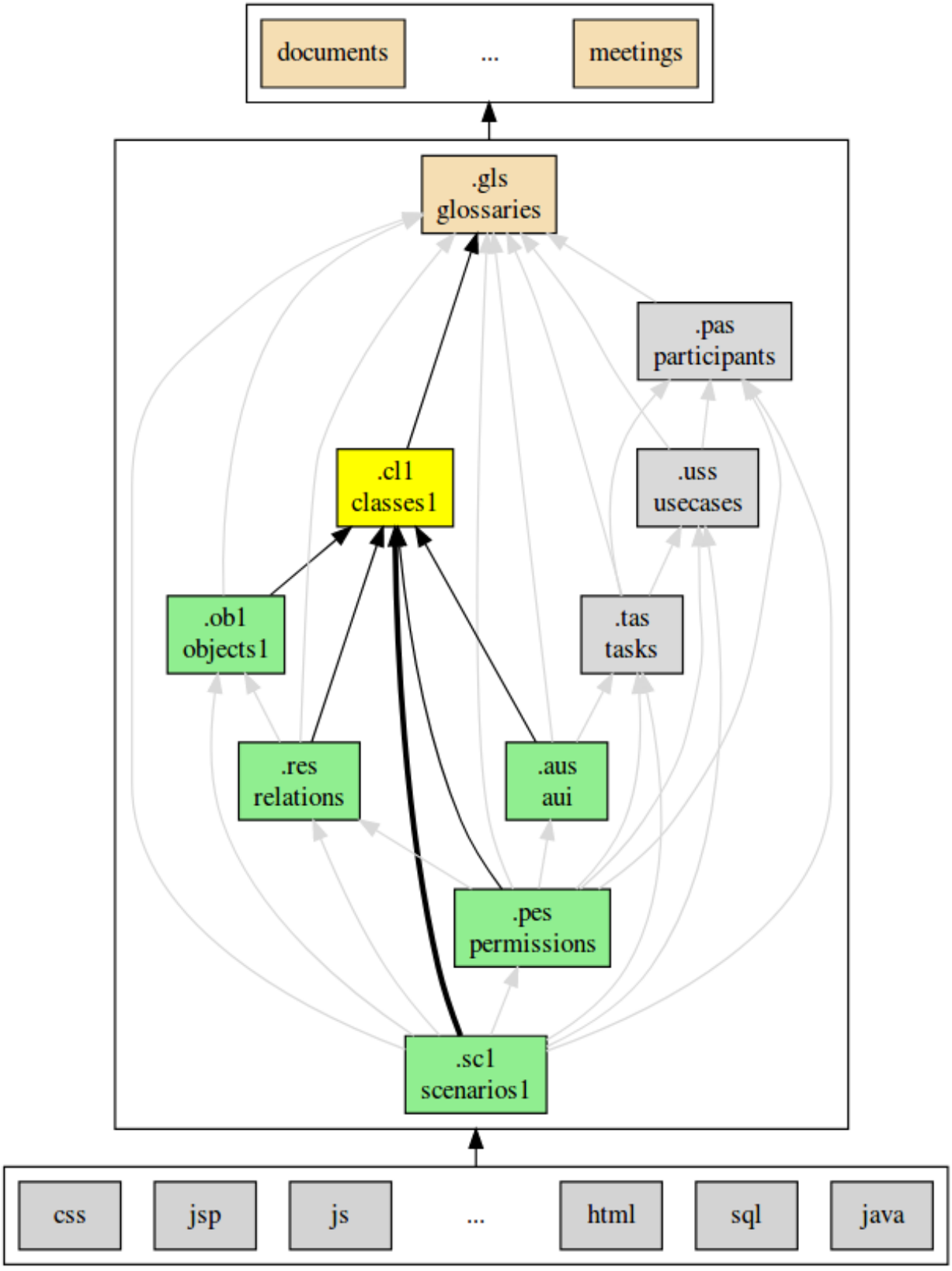
Par ailleurs lorsque plusieurs associations forment un cycle il assez probable qu'une ou des contraintes s'appliquent au sein de ce périmètre.

Constraintes OCL

Les contraintes exprimées en langage naturel (voir ci-dessus) peuvent ensuite être traduites en OCL en utilisant USE OCL

Dépendances

Le graphe ci-dessous montre les dépendances entre langages.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

2.1.4 ObjectScript1

Exemples

Le code ci-dessous montre un modèle d'objets basique :

```
! create bob : Personne
! bob.nom := 'bob'
! bob.age := 37
! insert(bob, c232) into EstResponsableDe
! create nourry : Enseignant
! nourry.nom := 'Nourry Blanc'
! nourry.matiere := 'musique'
! nourry.login := Undefined
! nourry.motDePasse := Undefined
! create s876 : Classe
! s876.code := 'S876'
! insert (nourry, s876) into IntervientDans
```

Le code ci-dessous montre un modèle d'objets annoté :

```
--| (1) Bob a 37 ans et est responsable de la classe c232
    ! create bob : Personne
    ! bob.nom := 'bob'
    ! bob.age := 37
    ! insert(bob, c232) into EstResponsableDe
--| (2) Nourry Blanc est professeur de musique.
    ! create nourry : Enseignant
    ! nourry.nom := 'Nourry Blanc'
    ! nourry.matiere := 'musique'
    ! nourry.login := Undefined
    ! nourry.motDePasse := Undefined
--| (3) Nourry Blanc intervient en terminale S876.
--| (4) Il a vraiment de la chance.
--| (5) La terminale S876 est plaisante.
    ! create s876 : Classe
    ! s876.code := 'S876'
    ! insert (nourry, s876) into IntervientDans
--| (6) Alicia Ganto est professeur de math.
```

Le code suivant montre un modèle d'objets négatif :

```
--@ violates EstResponsableDe.responsable.max
--@ violates ResponsableAdulte

--| (1) Bob a 30 ans et est responsable de la classe c232
    ! create bob : Personne
    ! bob.age := 30
    ! insert(bob, c232) into EstResponsableDe
--| (2) Octavia a 17 ans et est responsable la classe c232.
    ! create octavia : Personne
    ! octavia.age := 17
    ! insert(bob, c232) into EstResponsableDe
```

ObjectScript1

ObjectScript1 est une notation textuelle pour écrire des diagrammes d'objets UML. ObjectScript1 est une version réduite du langage SOIL (USE OCL). L'extension `.ob1` est utilisée à la place de l'extension `.soil`.

Concepts

Les modèles d'objets sont basés sur les concepts suivants :

- les valeurs d'énumérations,
- les objets,
- les valeurs d'attributs,
- les liens,
- les objets-liens,
- les textes annotés,
- les violations.

Valeur d'énumérations

ObjectScript1 (basé sur USE OCL):

```
Season::winter
```

Objets

ObjectScript1 (basé sur USE OCL):

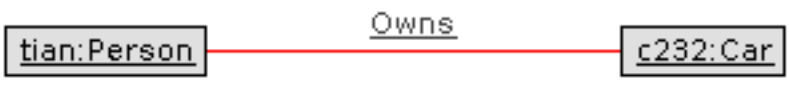
```
! create bob : Person
! bob.nom := 'bob'
! bob.dateDeNaissance := '21/10/1994'
```

Liens

ObjectScript1 (basé sur USE OCL):

```
! insert(tian,c232) into Owns
```

Diagramme d'objets UML :



Objet-liens

ObjectScript1 (basé sur USE OCL):

```
! c := new Hate between (chita,kaa)
! c.reason := "kaa is really mean"
! c.intensity = 1000
```

Textes annotés

ObjectScript1

```
--| Bob was born  ow
    ! create bob : Personne
    ! bob.nom := 'bob'
    ! insert(tian,c232) into Owns
--| (1) Nourry Blanc est professeur de musique.
    ! create nourry : Enseignant
    ! nourry.nom := 'Nourry Blanc'
    ! nourry.matiere := 'musique'
    ! nourry.login := Undefined
    ! nourry.motDePasse := Undefined
--| (2) Nourry Blanc intervient en terminale S876.
--| (3) Il a vraiment de la chance.
--| (4) La terminale S876 est plaisante.
    ! create s876 : Classe
    ! s876.code := 'S876'
    ! insert (nourry, s876) into IntervientDans
--| (3) Alicia Ganto est professeur de math.
```

Violations

Les violations sont des erreurs produites par un modèle d'objets appelé "modèle d'objets négatifs" (ou "modèle négatif d'objets"). Les violations sont déclarées à l'aide du mot clé `violates`. Il y a deux genres de violations ;

- **Violations de cardinalités**. Une telle violation se produit
  - soit lorsque la cardinalité effective associée à un role est supérieure à la cardinalité maximale déclarée,
  - soit lorsque la cardinalité effective est inférieure à la cardinalité minimale.

  Voici deux exemples possibles de violations :

  ```
  --@ violates EstResponsableDe.responsable.min
  --@ violates Dirige.directeur.max
  ```

  Dans cet exemple `EstResponsableDe` et `Dirige` sont des associations. `responsable`, `directeur` sont des rôles. `min` et `max` font référence à la cardinalité minimale et maximale associées aux rôles.

- **Violations de contraintes**. Ces violations se produisent lorsqu'un ou plusieurs objets violent une contrainte. Voici un exemple de contrainte de violations :

  ```
  --@ violates DirecteurAdulte
  ```

  Dans cet exemple `DirecteurAdulte` est une contrainte définie dans le modèle de classes.

  NOTE: les violations de contraintes ne sont détectées par l'outil USE OCL uniquement si la contrainte est définie en OCL.

Outils

Analyse des modèles d'objets

La conformité des modèles d'objets vis à vis du modèle de classes peut être vérifiée avec l'outil USE OCL. Lorsque la *méthode ModelScript* est utilisée entrer la commande suivante dans un terminal (on suppose que le répertoire courant est le répertoire racine du projet de modélisation) :

```
use -qv concepts/classes/classes.cl1 concepts/objets/o<N>/o<N>.ob1
```

L'analyseur vérifie qu'il n'y a pas d'erreurs de syntaxe, pas d'erreurs de type, pas d'erreurs de cardinalités et pas d'erreurs de contraintes. Si aucune erreur n'est affichée alors les deux modèles sont corrects et sont alignés.

Note: Si des violations sont définies (instructions `@violates`) le modèle d'objets doit produire les erreurs escomptées. Cette vérification n'est pas automatisée. Il faut donc vérifier "manuellement" que toutes les erreurs mentionnées sont effectivement produites.

La localisation des erreurs n'est parfois pas indiquée clairement. Si ce problème apparaît utiliser l'interpreteur USE en utilisant la commande suivante :

```
use -nogui concepts/classes/classes.cl1 concepts/objets/o<N>/o<N>.ob1
```

Si l'objectif est de vérifier les cardinalités utiliser ensuite la commande use `check` ou `check -v`. Terminer finalement avec la command `quit` ou `Ctrl C` pour sortir de l'interpréteur.

Génération de diagrammes

Créer des diagrammes d'objets est possible en utilisant l'outil USE OCL.

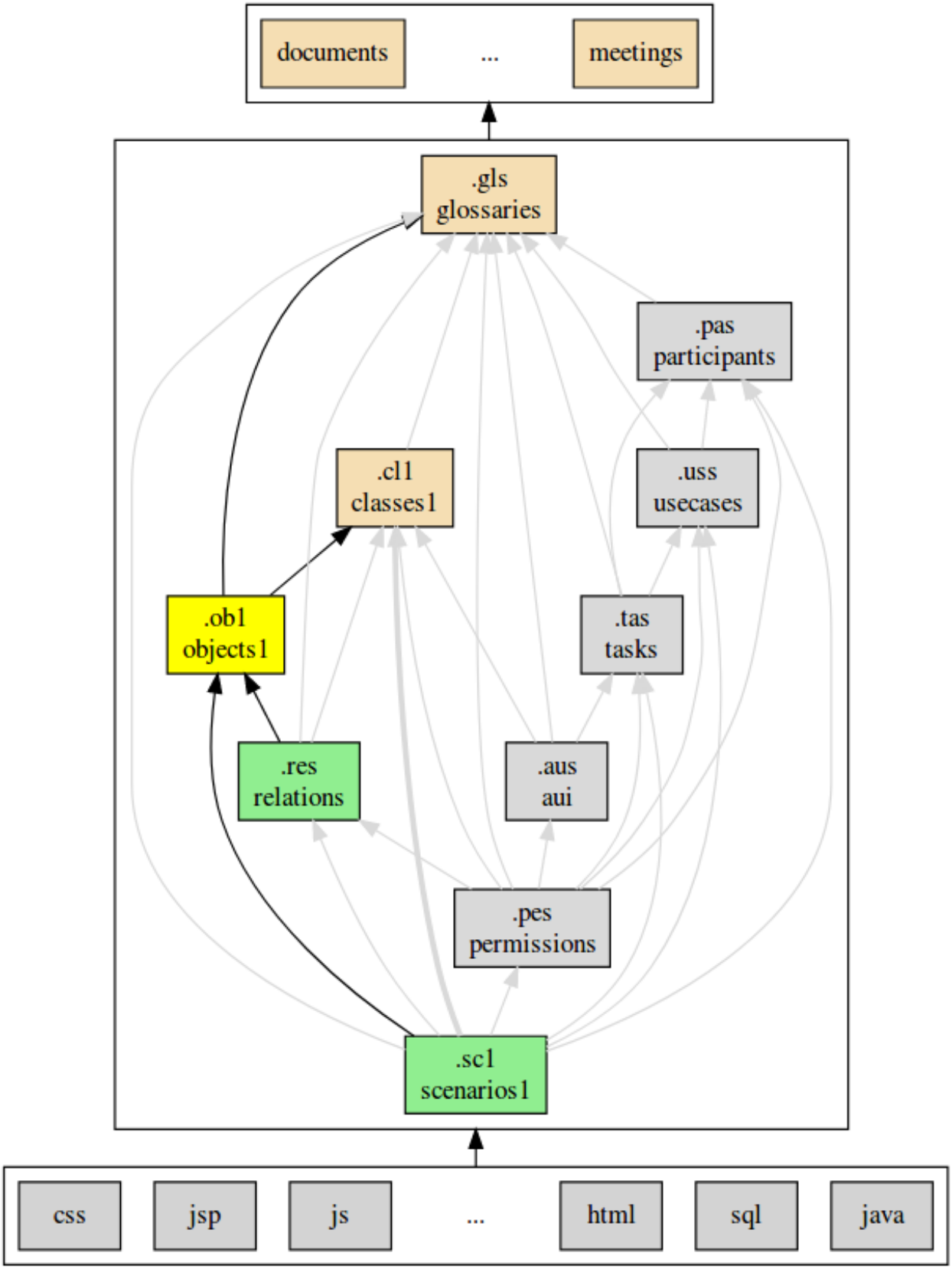
```
use -nr concepts/classes/classes.cl1 concepts/objets/o<N>/o<N>.ob1
```

Se référer à la page "creating UML object diagrams" pour plus d'information.

La disposition (layout) du diagramme doit être sauvé dans le fichier `concepts/objets/O<N>/diagrammes/o<N>.obd.clt`. Une copie d'écran doit être effectuée et sauvé dans `concepts/objets/O<N>/diagrammes/O<N>.obd.png`.

Dépendances

Le graphe ci-dessous montre les dépendances entre langages.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

2.1.5 RelationScript

Exemples

```
relation model CyberStore
import glossary model from '../glossaries/glossaries.gls'
import qa model from '../qa/relations.qas'

relation Employee(firstname_, salary, address, department)
    | All the employee in the store.
    intention
        (n, s, a, d) in Employee <=>
        | the `Employee` identified by her/his firstname <n>
        | earns <s> € per cycle. She/he lives
        | at the address <a> and works in the `Department` <d>.
    examples
        ('John', 120, 'Randwick', 'Toys')
    constraints
        dom(firstname) = String
        dom(salary) = Integer
        dom(address, department) = String
        key firstname
        firstname -> salary
        firstname -> address, department

relation Leaders(department_:String, boss:s)
    | The department leaders.
    intention
        (p, d) in Leaders <=>
        | The `Leader` of the `Department` <d> is the person <p>.
    constraints
        key department

relation Leaders2   # columns defined "vertically"
    | The department leaders.
    columns
        departement:String
        boss:String

constraints
    Leaders[department] = Employee[department]
    Leaders[boss] C= Employee[firstname]

constraint SalaryDifference
    | The difference of salary in a department must not exceed 100%.

dataset DS1
    | Employees and leaders of Alpha Super store.
    Employee
        ('John', 120, 'Randwick', 'Toys')
        ('Mary', 130, 'Wollongong', 'Furniture')
        ('Peter', 110, 'Randwick', 'Garden')
        ('Tom', 120, 'Botany Bay', 'Toys')
    Leaders
        ('John', 'Toys')
        ('Mary', 'Furniture')
        ('Peter', 'Garden')
```

(continues on next page)

(continued from previous page)

```
negative dataset NDS1
    | Octavia and bookstore do not exist.
    | Violation of referential integrity constraints.
    Employee
        ('John', 120, 'Randwick', 'Toys')
        ('Mary', 130, 'Wollongong', 'Furniture')
    Leaders
        ('Octavia', 'Bookstore')

query JohnBoss(boss)
    | The department leaders.
    (Employee:(firstname='John')[department] * Leaders)[boss]

view Salaries(name_:s, salary:i)
    | The salary of each employee.
    Employee[firstname, salary]

relation LesAppartements
    transformation
        from R_Class(Appartement)
        from R_Compo(EstDans)
        from R_OneToMany(Partage)
    columns
        nom_ : String
        numero_ : Integer
        superficie : Real
        nbDePieces : Integer
        jnum : Integer
    constraints
        key nom_, numero_
        LesAppartements[jnum] C= LesJardins[jnum_]
        LesAppartements[nom_] C= LesBatiments[nom_]
```

RelationScript

Le langage RelationScript permet d'exprimer des "schemas" au sens du modèle relationnel.

Note: Attention, le terme "modèle de relations" ne doit pas être confondu avec le terme modèle relationnel. Un modèle de relations permet de définir des relations, tout comme un modèle de cas d'utilisation définit des cas d'utilisation, un modèle de classes définit des classes, etc. Le modèle relationnel correspond au contraire à un concept plus général. Il s'agit d'une manière de structurer et d'interroger des données.

Concepts

Le langage RelationScript est basé sur les concepts suivants :

- les schémas, appelés modèles de relations, (relation models),
- les relations (relations),
- les colonnes (columns),
- les clés et les clés étrangères (keys et foreign keys),

- les contraintes (constraints),
- les dépendences fonctionnelles (functional dependencies),
- les formes normales (normal forms),
- les jeux de données (data sets),
- les requêtes (queries)
- les vues (views)
- les transformations.

Relations

Les relations peuvent être déclarées sur une seule ligne, en utilisant la notation simple que l'on trouve typiquement dans les livres ; par exemple :

```
R(x, y, z).
```

Il est également possible, et de manière tout à fait équivalente, de définir les colonnes de manière verticale (et optionellement d'ajouter le mot clé `relation`) :

```
relation R
    columns
        x
        y
        z
```

Clés

Dans les livres et par convention les attributs clés sont soulignés. En l'absence de soulignement des caractères ascii, en RelationScript le nom des attributs clés est suffixé par un caractère souligné "_".

```
R(x_, y_, z).
```

Dans l'exemple ci-dessus la clé est (x,y). Dans le cas où il y aurait plusieurs clés, les attributs peuvent être suffixés. Par exemple la relation suivante possède 3 clés :

```
R(x_id1, y_id2_id3, z_id3, t, u).
```

Telle qu'elle est définie la relation possède 3 clés : < (x), (y), (y,z) >. Dans tous les cas les clés peuvent être spécifées de manière plus explicites dans la section `constraints`.

```
relation R(x, y, z)
    constraints
        key x
        key y
        key y,z
```

Intention

L'intention d'une relation correspond à sa signification, à la manière d'interpréter le contenu d'une relation. L'intention peut soit être implicite, soit de être définie de manière explicite et structurée. Dans l'exemple ci-dessous l'intention est implicite, la relation est définie sous forme de documentation non structurée.

```
relation R4(a_,c,d)
    | The list of X. This relation means that ...
```

Il est préférable de définir l'intention de manière structurée comme ci-dessous. Notons que `dans` est un mot-clé (`in` en anglais) et que la ligne correspondante à une structure. Le nombre de paramètres du tuple doit correspondre au nombre d'attributs de la relation. Dans le texte de l'intentation les variables doivent apparaître entre crochets (p.e. `<a>`)

```
relation R4(a_,c,d)
    | The list of X.
    intention
        (a,c,d) in R4 <=>
        | the person <a> is ... with <c> ... and <d> ...
```

Contraintes de domaine

Le domaine des attributs peut être défini de différentes manière comme le montre les exemples suivants :

```
relation R1(a,b,c,d)
    constraints
        dom(a) = String
        dom(b) = dom(c) = Date
        dom(d) = Real ?

relation R2(a:String, b:Date, c:Date, d:Real ?)

relation R3
    columns
        a : String
        b : Date
        c : Date
        d : Real ?
```

Un type basique suivi de de l'opérateur `?` signifie que le domaine est étendu avec la valeur `null`. En d'autres termes cela signifie que l'attribut correspondant est optionnel.

Note: Le modèle relationnel n'autorise pas les attributs optionnels. Ces cette possibilité est offerte pour faciliter la traduction vers SQL.

Différents types de données sont définis par le langage RelationalScript. Chaque type de données possède sa propre notation abbréviée, ce qui s'avère pratique lors de la définition de relations sur une seule ligne.

Datatype	Shortcut
String	s
Real	r
Boolean	b
Integer	i
Date	d
DateTime	dt
Time	t

En utilisant la notation abbréviée une relation peut être définie comme suit :

```
relation LesEmployés(nom:s, prenom:s, age:i, dateNaissance: d)
```

Contraintes d'intégrité

Les contraintes d'intégrité (et en particulier les contraintes d'intégrité référentielle) peuvent être définies sous forme de documentation en langue naturelle.

```
constraint Parent
    | Les parents d'une personne doivent être
    | plus agés que cette personne, d'au moins 7 ans.
```

Si le modèle de relations est dérivé d'un modèle de classes, il n'est pas nécessaire de répeter le corps des contraintes qui sont simplement "héritées" ; seul le nom suffit.

```
constraint AuMoins7Ans
```

Le corps de certaines contraintes peut également être défini en utilisant l'*algèbre relationnelle*.

```
constraint FK_34h
    | The h of the relation R3 is one of the h of R4.
    R3[h] C= R4[h]

constraints
    R1[d] C= R2[d]
    R1[d1,d1] C= R2[d1,d2]
    R[X] u R[z] = {}
    R[X] n R[z] = Persons[X]
```

Voir la section concernant l'*algèbre relationnelle* pour plus de détails sur la notation utilisée.

Dépendences fonctionnelles

Les dépendances fonctionnelles et les concepts associés peuvent être définis comme suit :

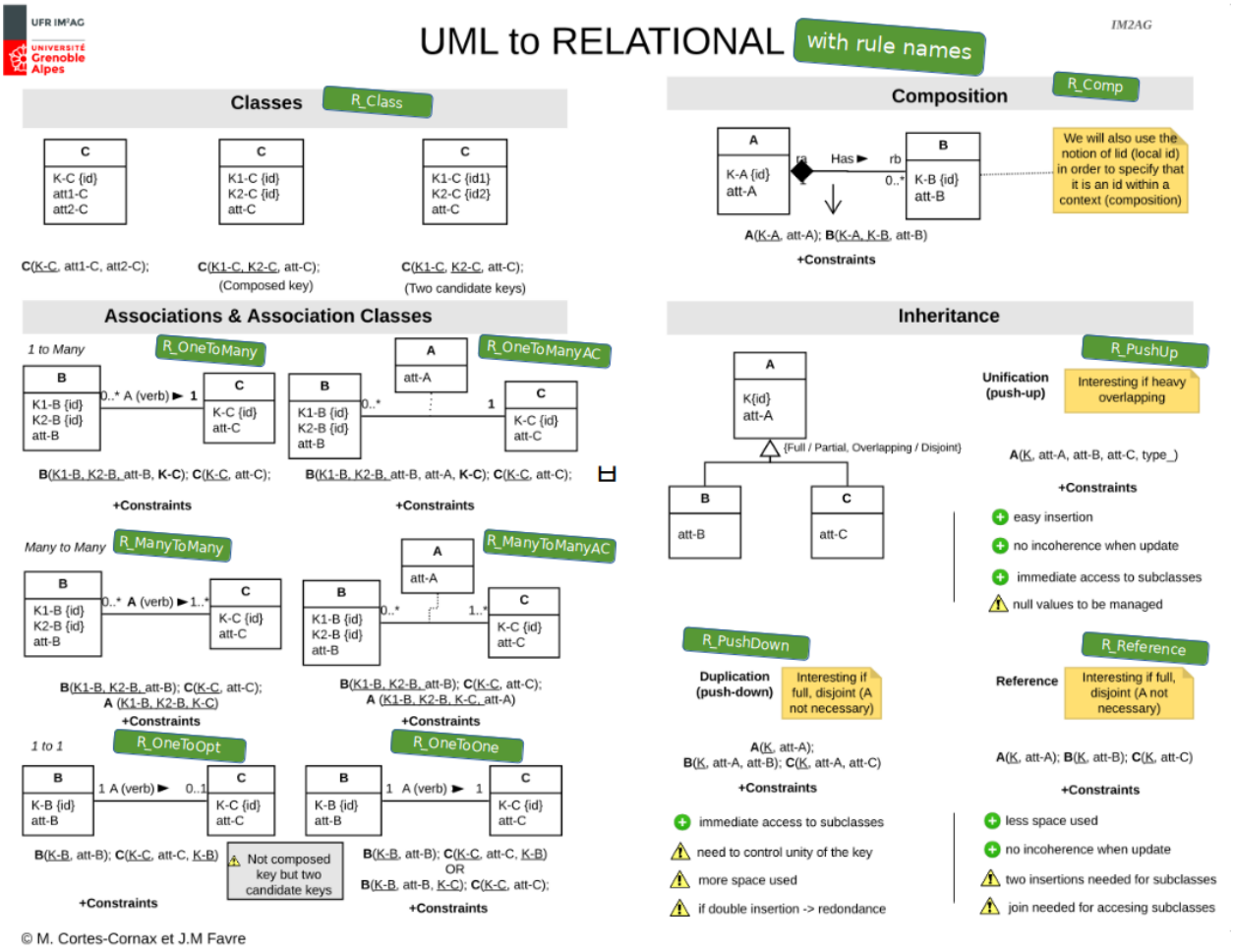
```
relation R(a,b,c,d)
    constraints
        key a,b
        a,b -> c,d
        prime a
        prime b
        /prime c
        a -/> c
        c -ffd> d
        a -/ffd> b
        {a}+ = {a,b,c}
```

Formes normales

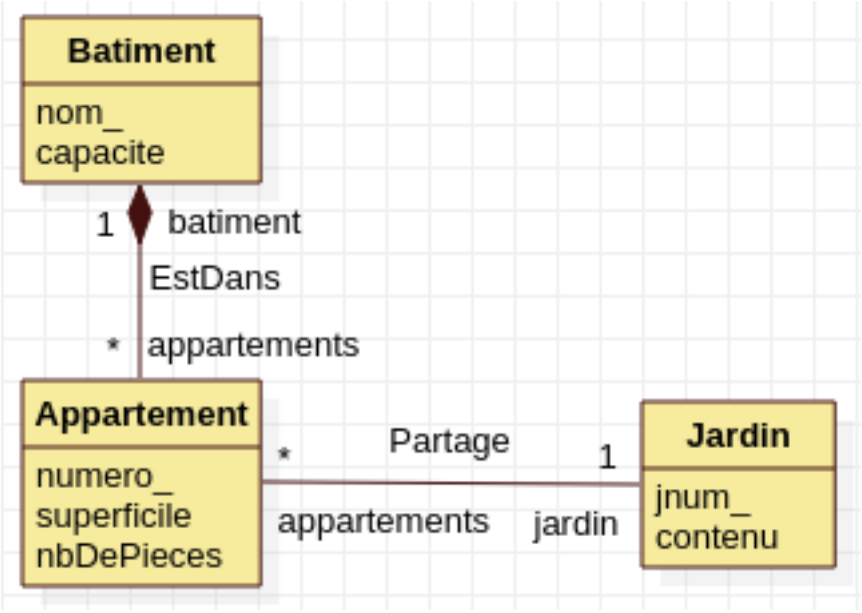
```
relation R(a,b,c,d)
    constraints
        3NF
```

Transformations

Un modèle de relations peut être obtenu par transformation à partir d'un modèle de classes. Une telle transformation peut correspondre à l'application d'une suite de règles définies dans une catalogue. Un exemple de catalogue est montré ci-dessous à titre d'illustration (`télécharger`).



Considérons de plus le modèle de classes ci-dessous.



En utilisant la règle `R_Class` la classe `Batiment` est transformée en la relation `LesBatiments` définie comme suit :

Note: La version française de la règle fait le renommage suivant : `X` devient `LesXs`.

```
relation LesBatiments
    transformation
        from R_Class(Batiment)
    columns
        nom_ : String
        capacite : Integer
    constraints
        key nom_
```

Remarquer la section `transformation` et le mot clé `from`. Vient ensuite le nom de la règle utilisée `R_Class` qui ici est appliquée à la classe `Batiment`. Le reste de la définition de la relation est standard.

La transformation de la classe `Appartement` est montrée ci-dessous à titre d'illustration.

```
relation LesAppartements
    transformation
        from R_Class(Appartement)
        from R_Compo(EstDans)
        from R_OneToMany(Partage)
    columns
        nom_ : String
        numero_ : Integer
        superficie : Real
        nbDePieces : Integer
        jnum : Integer
    constraints
        key nom_, numero_
        LesAppartements[jnum] C= LesJardins[jnum_]
        LesAppartements[nom_] C= LesBatiments[nom_]
```

Dans la section `transformation` ont voit que trois règles ont été appliquées :

- la règle de transformation de classe `R_Class`,
- la règle de transformation de composition `R_Compo`. Cette règle a été appliquée à la composition `EstDans`,
- la règle `R_OneToMany` appliquée à l'association `Partage`.

Dans certains cas il est nécessaire de changer le nom de certains attributs, de fusionner deux attributs en une même colonne, etc. Certaines règles peuvent être manquantes ou doivent être appliquée de manière différente. Toutes ces modifications peut être documentées sous forme de documentation dans la section transformation.

```
relation LesX
    transformation
        from R1(X)
        | Le type de l'attribut z a été changé vers String car ...
        | L'attribut u a été préfixé par le nom du rôle car ...
        | ...
```

Requêtes

Les requêtes sont simplement des relations dont le corps est exprimé à l'aide de l'*algèbre relationnelle*.

```
query Q1(boss)
    | The department leaders
    (Employe:(firstname='John')[department] * Leaders)[boss]
```

Vues

Au niveau du modèle relationnel les requêtes et les vues sont en tout point équivalentes. Le concept de vue est défini ici pour simplifier la transformation vers le langage SQL.

```
view V1(boss)
    | The department leaders
    (Employe:(firstname='John')[department] * Leaders)[boss]
```

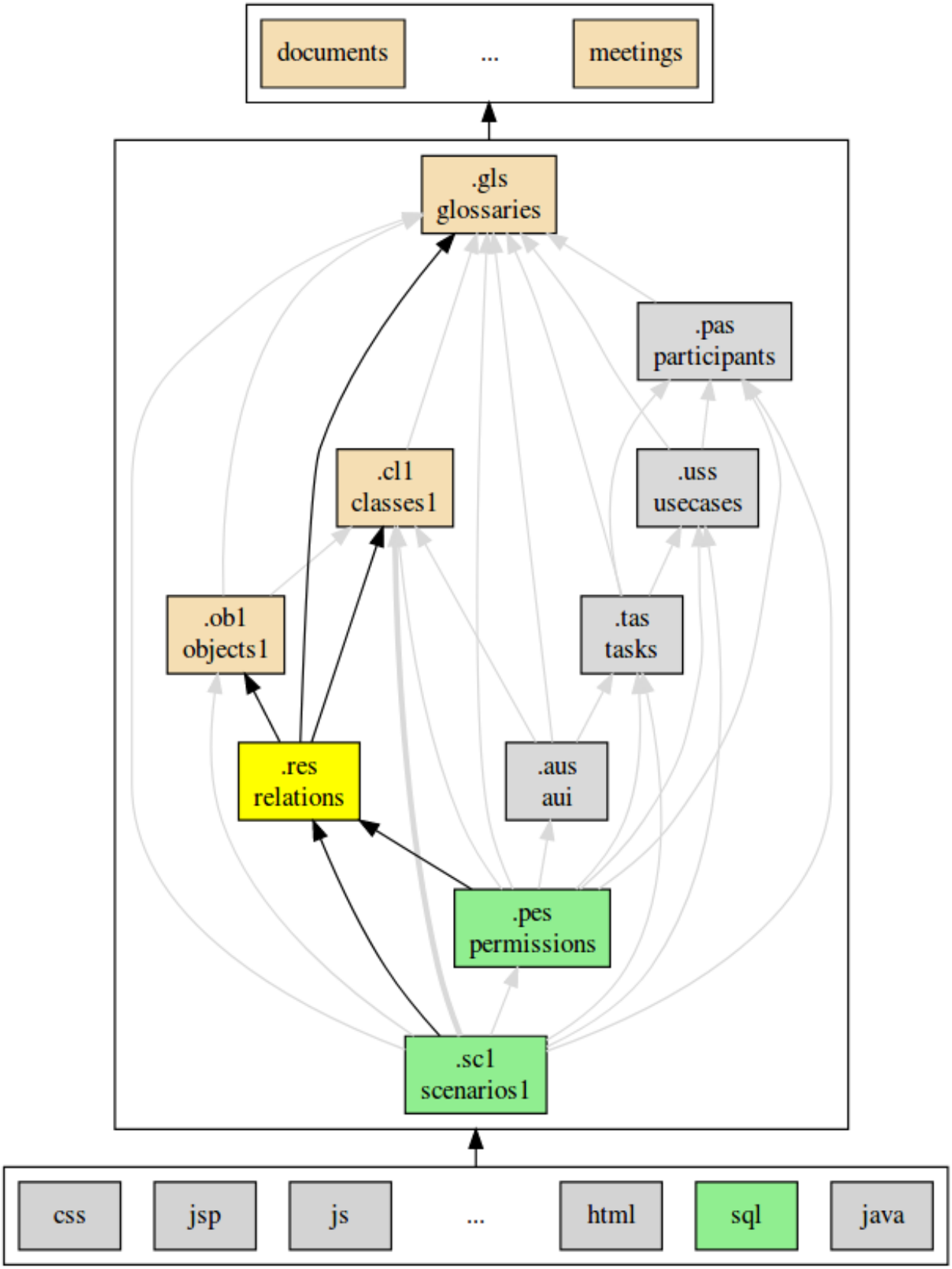
Algèbre relationnelle

Le langage RelationScript définit tous les opérateurs classiques de l'algèbre relationnelle (wikipedia). A chaque opérateur est associé une notation en ascii.

Operateur	Exemple
Projection	Employee[salary]
Selection	Employee :(address='Randwick')
Renaming	L(employee, address) := Employee[firstname, address]
Cartesian product	Employee x Leaders
θ join	Employee * (Employee.dept=Leaders.dept) Leaders
Natural join	Employee * Leaders
Union	Employee[firstname] u Leaders[firstname]
Intersection	Employee[firstname] n Leaders[firstname]
Difference	Employee[firstname] - Leaders[firstname]
Empty set	{}
Set inclusion	Employee C= Person
Set inclusion	Employee C Person
Set equality	Employee = Person
Intersection	Employee n Person
Union	Employee u Person
Tuple	(10, 3, 'Hello)

Dépendances

Le graphe ci-dessous montre les dépendances entre langages avec un focus sur le langage RelationScript.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

2.1.6 ParticipantScript

Exemples

```
participant model Demo

import glossary model from '../glossaries/glossaries.gls'

//=============================================================================
// "Class" level participants
//-----------------------------------------------------------------------------
// "Actors" are defined by UML usecase model ; they represent (classes of) users.
// "Stakeholders" have some interest in the system and/or its development.
// "Team roles" collaborate to design and develop the system/
//=============================================================================


//--- actors ------------------------------------------------------------------

actor Cashier
    | Cashiers are employee of `Cinemas`. The role of `Cashiers`
    | is to sell `Ticket` to `Spectators`. They also manage
    | `Subscriptions`. To perform these tasks `Cashiers` should have a
    | desktop application at their disposal.

actor HighCashier < Cashier
    | HighCashiers can cancel `Transactions` and launch
    | `MoneyBack` operations.

actor Client
    | Clients are people that interact with the web interface
    | of the system or that take their `Ticket` at a
    | `VendingMaching`. Most of them do not know the system,
    | or experienced have less than




//--- stakeholders ------------------------------------------------------

stakeholder role Treasurer
    | The role of treasurers is to check that all `FinancialTransactionq`
    | processed by the system are accurate.

stakeholder role SecurityManager
    | The role of the SecurityManager is to ensure the security in all
    | `Cinemas` and in particular in all `Rooms`. It should be possible
    | for example to inform SecurityManagers when an accident occur
    | in some `Room` or when a `Cinema` is overcrowded.




//--- team roles --------------------------------------------------------

team role Developer
    | A developer is responsible to design, develop, test and
```

(continues on next page)

(continued from previous page)

```
    | maintain models and pieces of code.

team role QualityManager
    | The QualityManager is responsible to define, with other
    | members of the development team, `QA` standard.
    | She also monitors `QC` process although she can to delegate
    | actual controls to other team members.
    |
team role QualityMaster < QualityManager
    | A `QualityMaster` has all duties and privileges of
    |`QualityManager` but she also has the power to change
    | the content of `QA` and `QC` standard.

team role ScrumMaster
    | The `ScrumMaster` is the team role responsible for
    | ensuring the team lives agile values and principles and
    | follows the processes and practices that the team
    | agreed they would use.
    | The responsibilities of this role include:
    | * clearing obstacles,
    | * Establishing an environment where the team can be effective
    | * Addressing team dynamics
    | * Ensuring a good relationship between the team and
    |   product owner as well as others outside the team
    | Protecting the team from outside interruptions and distractions.

team role ProductOwner
    | The `ProductOwner` responsibility is to have a vision of
    | what she wishes to build, and convey that vision to the
    | `ScrumTeam`.


//================================================================================
//   "Instance" level participants
//--------------------------------------------------------------------------------
// Both personae and persons are at the instance level: they belong to
// one of many participant class (actor, stakeholder or team role)
// Personae are fictional characters that serve as instance of actors.
// Persons are real-life people.
//================================================================================


person marieDupont : Developer, QualityManager
    name : "Marie Dupont Laurent"
    trigram : MDL
    portrait : './mdupont.png'


persona marco : Cashier, Client
    name : "Marco Gonzales"
    trigram : MGS
    portrait : './mdupont.png'
    | Marco is 45 years old.
    | He is used to computers and phones.
    | Some more description about marco
    attitudes
        | marco likes playing football.
```

(continues on next page)

(continued from previous page)

```
        | He also loves eating pizza and playing with this
        | damned computer system.
    aptitudes
        education
            | master software engineering (1992)
            | PhD in medio chemicals (1999)
        languages
            | english (fluent)
            | spanish (novice)
        age : 45
        disabilities : "blind"
        learning ability : low
        | Marco is kind to learn but he also knows already
        | very much.
    motivations
        why
            | Marco is really reluctant to use the system.
            | Her boss, anna, told him that he will be fired
            | if he do not get good results.
        level : low
        kind : obliged
        | Some additional remark or documentation
    skills
        | Marco is an expert in playing with the mouse.
        level : novice
        culture
            | occidental
        modalities
            "labtop" : expert
            "smartphone" : novice
            "iPhone 10.3" : expert
        environments
            "Ubuntu" : expert
            "Windows" : intermediate
            "Android 18.5" : novice


adhoc persona jean : Cashier, Client
    | Jean is 50 years old.
```

ParticipantScript

Le modèle de participants a pour but de définir les différents types de participants impliqués d'une manière ou d'une autre dans le projet et le logiciel. Cela peut être soit parcequ'un participant utilise le logiciel soit parcequ'il est impliqué dans sa conception.

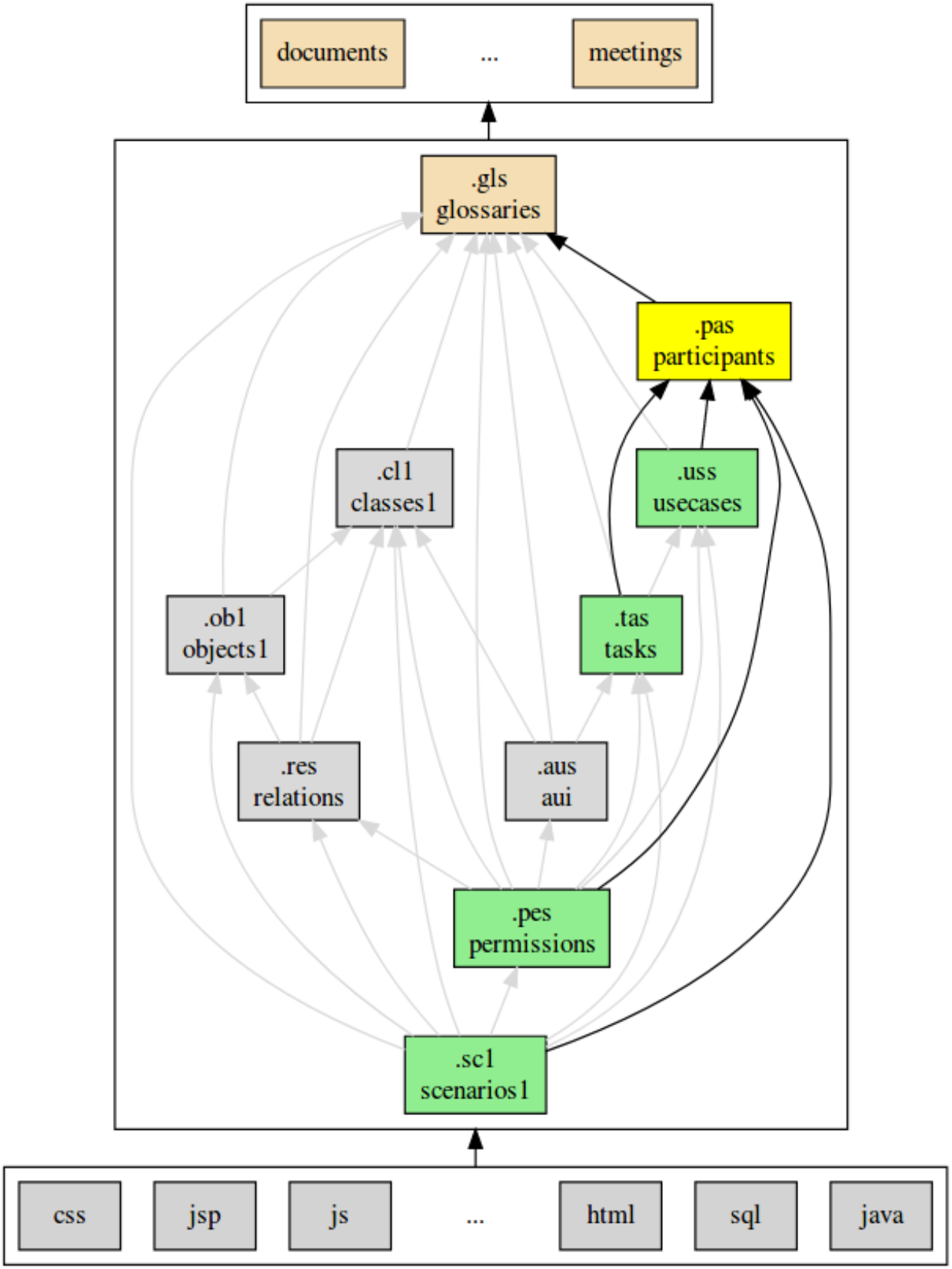
Concepts

- les acteurs,
- les parties prenantes,
- les rôle,
- les personnes,

- les personnages.

Dependances

Le graphe ci-dessous montre les dépendances entre langages.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

2.1.7 UsecaseScript

Exemples

Un **modèle préliminaire de cas d'utilisation** pourrait ressembler à l'exemple ci-dessous :

```
usecase model CyberDepartment

import glossary model from '../gls/glossaries.gls'
import usecase model from '../uss/usecases.uss'

                            // actors imported from the usecase model:
                            //    CEO, Employee, Manager, Secretary

interactions
    a CEO can CreateADepartment
    a Secretary can CreateADepartment
    a Secretary can AddAnEmployee
    a CEO can BrowseTheBudget
    a Manager can SetTheBudget
    an Employee can BrowseADepartment

usecase BrowseTheBudget
    actor CEO
    | The `CEO` want to see the performance of
    | each `Department` and make sure that
    | each `Budget` allocated is sufficient.

usecase AddAnEmployee
    actor Secretary
    | A `Secretary` add a new `Employee` into
    | the system and assign this `Employee` to
    | her `PrimaryDepartment` in order to
    | sure that the `ProvisionalBudget`
    | will be enough. The employee is validated
    | only after `SetAnEmployee` is performed.

usecase SetAnEmployee
    actor Manager
    | A `Manager` can confirm the `Position`
    | and `Salary` of an `Employee` already
    | added in the system.

...
```

Un **modèle détaillé de cas d'utilisation** pourrait ressembler à l'exemple ci-dessous.

```
...

usecase CreateADepartment
    | Very short summary of the usecase.
    primary actor CEO
    secondary actor Secretary
    persona Toufik
        | Toufik is responsible for most of the department creation.
        | He perform this usecase without the help of anyone.
        volume
```

(continues on next page)

(continued from previous page)

```
        | 3 days of work
        | 100 units to define
    frequency
        | more than 1 creation per year
persona Celia
    | Celia back up toufik when he is traveling or at the end
    | of the year when he is very busy. She is
    frequency
        | less than 1 creation for 5 year
description
    | This description is longer than the summary,
    | yet less structured than the "flow" of events.
    | To be used where appropriate.
goal
    | This section describes the goals of the actor(s).
    | What they try to acheive by performing the usecase.
    | This section is useful to make sure that the usecase
    | has a real business value. So-called "essential
    | usecases" are based on this information.
precondition
    | The condition that is necessary for the usecase to
    | be performed. When the condition is satisfied the
    | usecase could be executed, but only if the "trigger"
    | (see below) is activated
trigger
    | The event that make the usecase start.
postcondition
    | The condition that is satisfied at the end of the
    | execution of the usecase.
risk: low
    | The risk associated with the implementation of the
    | usecase.
frequency
    | The estimate about the usecase frequency.
    | This could be for instance "twice a year", "10 per hour".
volume
    | The estimate about the volume of data to be processed
    | for example. This could be something like '100 units to
    | be created in average".
flow
    | The flow of events describing the "nominal flow",
    | that is the most important/common scenario.
    | The flow should be defined as a sequence of step,
    | each step being prefixed by a number between parenthesis.
    | For instance:
    |
    | (1) first step.
    | (2) second step. The description of this step does not fit
    |     in one line so it is indented.
    |     Yet another line in the description of step (2).
    | (3) third step
    | ...
    |
extension EmployeeAlreadyDefined at step 2
    when
        | When this condition is satisfied in step 2 of the normal
        | flow then this extension is executed.
```

(continues on next page)

(continued from previous page)

```
        flow
            | The alternate flow for this extension.
            | (1) step 1 for this extension.
            | ...
            | (n) return to CreateDepartment.4
        usecase RemoveAnEmployeeOccurrence


...
```

UsecaseScript

UsecaseScript permet le développement incrémental des modèles de cas d'utilisation avec en particulier :

- les **modèles préliminaires**. Les cas d'utilisation sont décrits de manière synthétique.
- les **modèles détaillés**. La description des cas d'utilisation est complète, avec pour chaque cas d'utilisation des scénarios d'exécution.

Notons que dans tous les cas les acteurs et les personnages sont décrits (de manière plus ou moins détaillée) dans le modèle de participants.

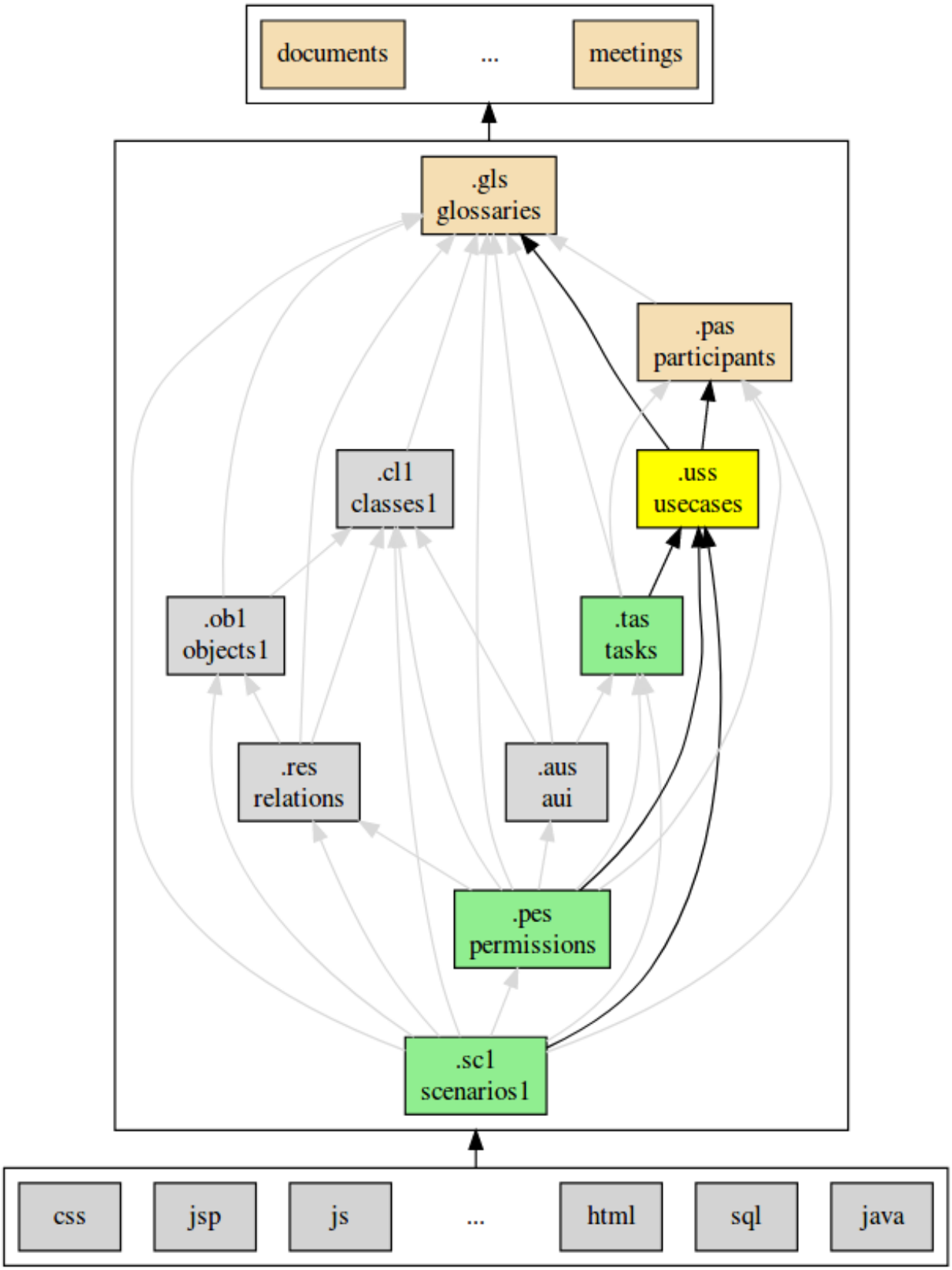
Concepts

Les modèles de cas d'utilisation sont basés sur les concepts suivants :

- les **acteurs**. Ils sont en principe définis dans les modèles de participants.
- les **cas d'utilisation**.
- les **interactions**.

Dépendances

Le graphe ci-dessous montre les différentes dépendances entre langages et en particulier celles reliées au langage UsecaseScript.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

2.1.8 TaskScript

Examples

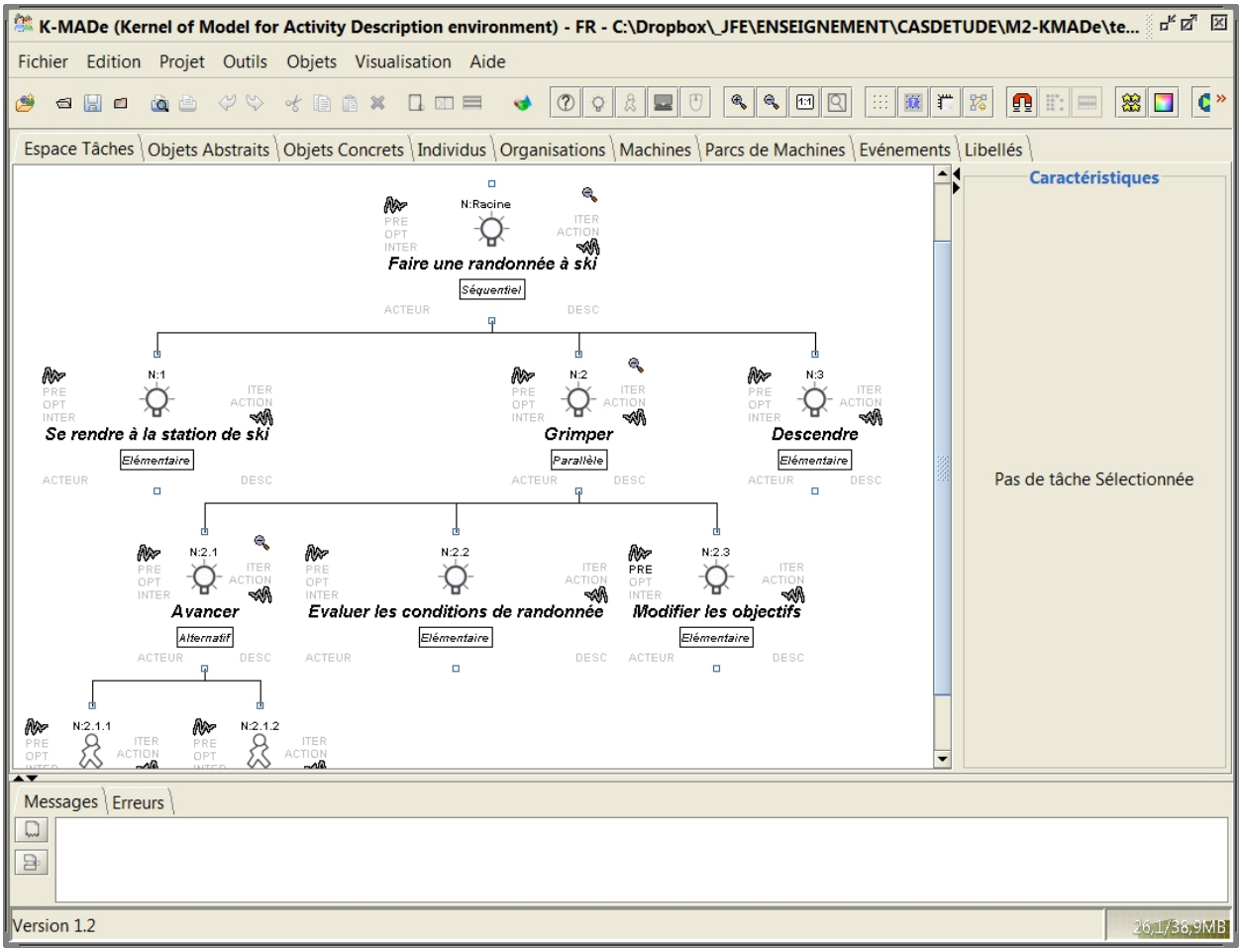


Fig. 1: *task model* by Sybille Caffiau

TaskScript

Tasks models exist in the form of *task diagrams* created and managed using the Kmade tool. By contrast to other languages, no textual notation is available for *task models*.

For more information about Kmade see the Kmade user manual.

Note that the integration between Kmade and ModelScript is achieved via the XML representation of Kmade model.

Concepts

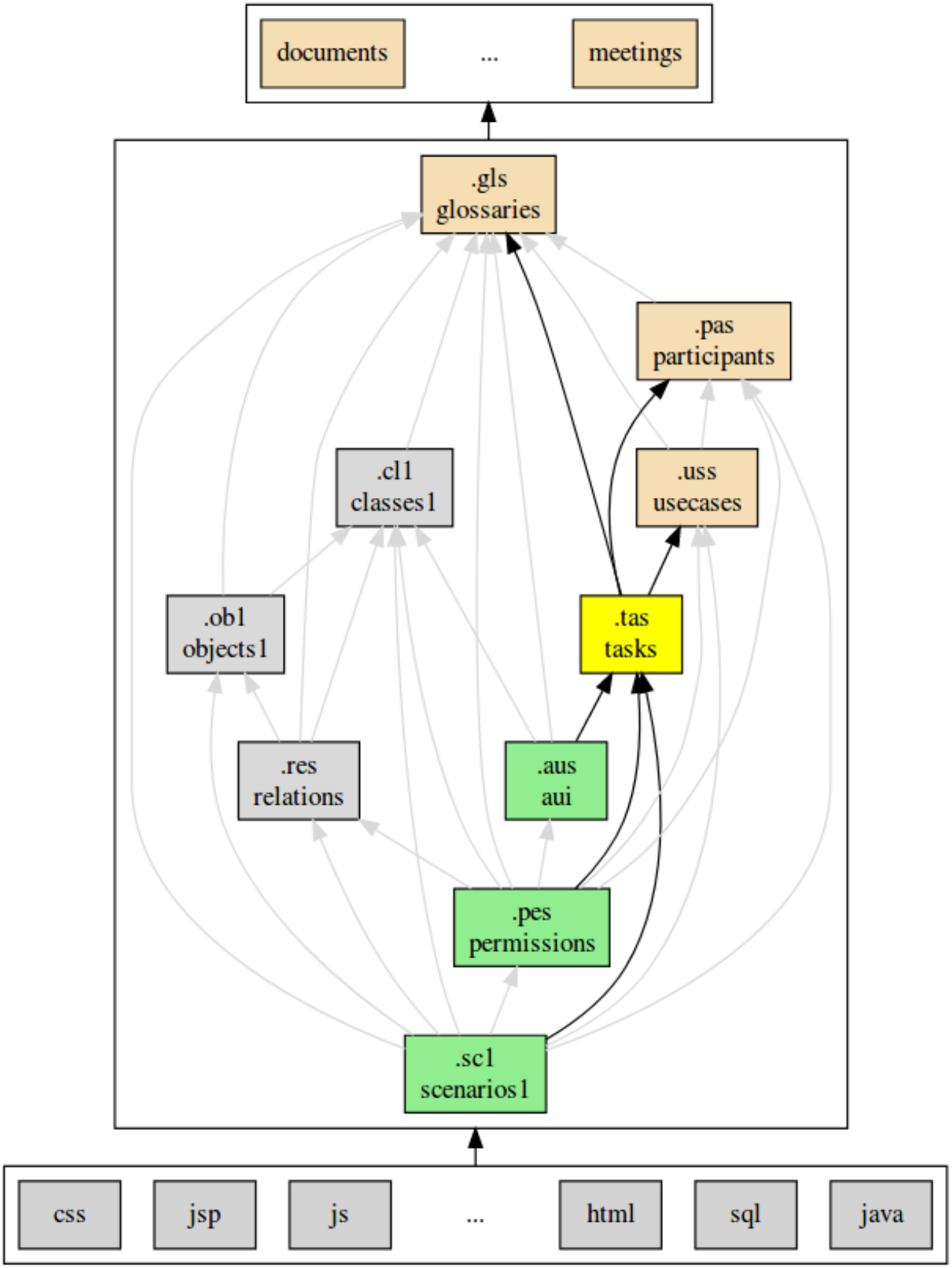
TaskScript is based on the following concepts:

- tasks
- task trees

- task decompositions
- concepts

Dependencies

The graph below shows all language dependencies.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

2.1.9 AUIScript

Warning: At the time being abstract user interfaces (AUI) are currently to be described only informally using a "paper and pencil" method. This page present a candidate language to represent AUI more formally. It is **not** to be used in current projects. It is shown here just to show how abstract user interface modeling could be integrated in ModelScript. The language is presented as-is without any guarantee that it fits expert needs.

Examples

```
aui model Demo

space EntrerLesInformations
    | Some documentation
    concepts
        email
        numerotel
    links
        ChoisirTypeReservation
        EntrerLesInformations


space EntrerLesInformations "Réservation"
    concepts
        email "email"
        numerotel "numéro de téléphone"
    links
        ChoisirTypeReservation "type"
        EntrerLesInformations "détail"
        back to EntrerLesInformations "précédent"
    transformation
        from
            Informer
        rule R1
        rule R2
        | Some explainations

space ChoisirTypeReservation
    links
        ReservationSansPayer
        Reserver
        back to EntrerLesInformations

space ReservationSansPayer
    links
        back to ChoisirTypeReservation
        PreciserCriteresDeRecherche

space PreciserCriteresDeRecherche
    links
        EntrerLesInformations

space Reserver
    links
        back to ChoisirTypeReservation
        Payer
```

(continues on next page)

(continued from previous page)

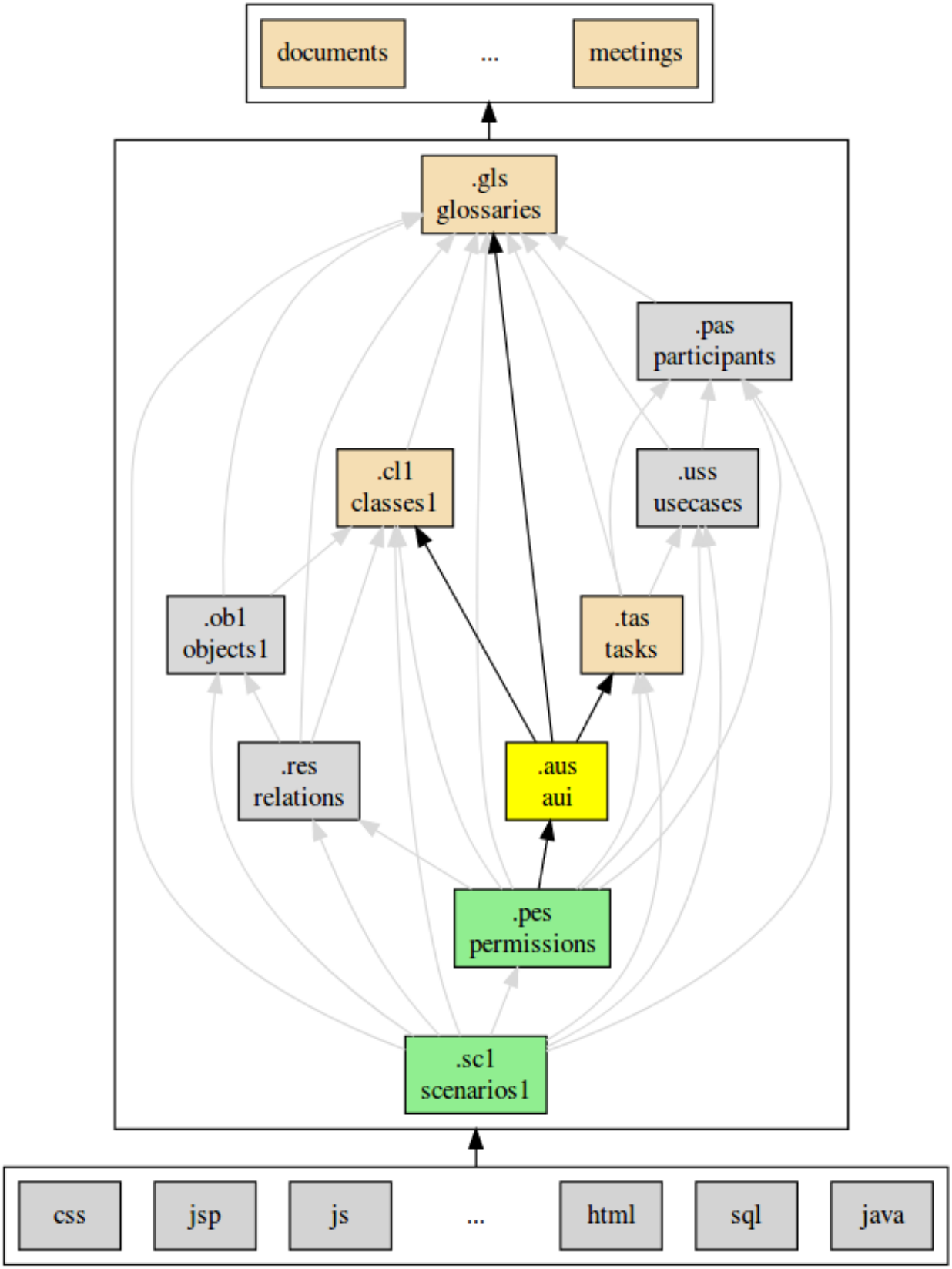
```
space Payer
    concepts
        modeDePaimement
        numeroDeCarte
    links
        ChoisirTypeDeBillet
        Payer

space ChoisirTypeDeBillet
    concepts
        pdf
        mobile
    links
        back to PreciserCriteresDeRecherche


space PreciserCriteresDeRecherche
    links
        EntrerLesInformations
```

Concepts

- spaces
- concepts
- links
- transformations

Dependencies

The graph below show all language depdencies.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

Spaces

```
space EntrerLesInformations "Réservation"
    | Some documentation
```

In the example above it is specified that “Réservation” can be used in the concrete user interface.

The space Space can contains concepts, links and transformations.

Concepts

```
space EntrerLesInformations "Réservation"
    concepts
        email "email"
        numerotel "numéro de téléphone"
```

Links

```
space ReservationSansPayer
    links
        back to ChoisirTypeReservation
        PreciserCriteresDeRecherche "Filtrer"
```

Transformation

```
space EntrerLesInformations "Réservation"
    transformation
        from
            Informer
        rule R1
        rule R2
        | Some explanations
```

It is possible to document from which tasks a given space comes from. Applied rules can be specified and additional explanations can be added.

2.1.10 ScenarioScript1

Exemples

Le développement de scénarios peut se faire en 3 étapes :

- (1) développement des **scénarios textuels** (phrases),
- (2) développement des **scénarios états** (phrase+instructions),
- (3) développement des **scénarios cas d’utilisation** (phrase+instructions+blocs).

Scénarios textuels

Les scénarios textuels sont des suites de **(phrases)**.

```
--@ scenario model S1
--@ import glossary model from "../../glossaries/glossary.gls"

--| phrase1
--| phrase2
--| phrase3
--| phrase4
--| phrase5
--| phrase6
--| phrase7
--| phrase8
```

Scénarios états

Les scénarios états sont caractérisés par l'état qu'ils font évoluer. Concrètemment les scénarios états sont constitués de phrases annotées par des instructions. Ces instructions correspondent à la l'évolution au cours du temps d'un modèle d'objets. Les scénarios états permettent de répondre à la question "comment l'état du système évolue ?".

```
--@ scenario model S1
--@ import glossary model from "../../glossaries/glossary.gls"
--@ import class model from "../../classes/classes.cls"

--| phrase1
--| phrase2
    ! instruction1
    ! instruction2
--| phrase3
    ! instruction3
    ! instruction4
--| phrase4
--| phrase5
--| phrase6
    ! instruction5
    ! instruction6
    ! instruction7
--| phrase7
    ! instruction8
--| phrase8
```

Comme le montre l'exemple ci-dessus certaines phrases peuvent ne correspondre à aucun changement d'état.

Scénarios cas d'utilisation

Les scénarios cas d'utilisation définissent "l'empreinte" des cas d'utilisation sur les scénarios états. Chaque phrase/instruction est positionnée par rapport au cas d'utilisation en cours. Alors que les scénarios états permettent de répondre à la question "comment le système évolue" les scénarios cas d'utilisation permettent de répondre à la question "qui fait quoi et pourquoi ?".

```
--@ scenario model S1
--@ import glossary model from "../../glossaries/glossary.gls"
```

(continues on next page)

(continued from previous page)

```
--@ import class model from "../../classes/classes.cls"
--@ import participant model from "../../participants.pas"
--@ import usecase model from "../../usecases/usecases.uss"

--@ context
    --| phrase3 (modifiée)
        ! instruction3
        ! instruction4

--@ personnage1 va usecase1
    --| phrase1
    --| phrase2
        ! instruction1
        ! instruction2

--| phrase4 (modifiée)
--| phrase5

--@ personage2 va usecase2
    --| phrase6
        ! instruction5
        ! instruction6
        ! instruction7
    --| phrase7
        ! instruction8

--| phrase8
```

Les blocs `context` correspondent au contexte du scénarios, c'est à dire à la construction de l'état initial. Ils s'agit de la modèlisation de l'ensemble des informations existant avant que le scénario démarre.

Considérons un exemple où la phrase `tim a 15 ans` est suivie de l'instruction `tim.age := 15`. Première possibilité, la plus probable, ces deux instructions font a priori partie du contexte. Autre solution, ces informations font partie d'un bloc cas d'utilisation si `tim` change d'age durant le scénario (peu probable, mais cela dépend du scénario).

Bien évidemment dans les deux cas on suppose que l'age de tim doit être modélisé pour le bon déroulement du scénario. Si ce n'est pas le cas l'instruction `tim.age := 15` doit être éliminée et la phrase `tim a 15 ans` doit être en dehors de tout bloc, au premier niveau. Cette information est peut être importante pour le scénario, même si elle n'a pas d'impact directe sur l'état du système.

Outils

Analyse de modèles

La conformité des modèles de scénarios par rapport au modèle de classes peut être verifiée par l'outil USE OCL avec la même procédure que pour *l'analyse des modèles d'objets*.

Note: ATTENTION, la conformité avec le modèle de cas d'utilisation n'est pas vérifiée.

Génération de diagrammes

Il est possible de générer un diagramme d'objets correspondant à l'état final du scénario. Utiliser pour cela la même procédure que pour *génerer un diagramme d'objet standard*.

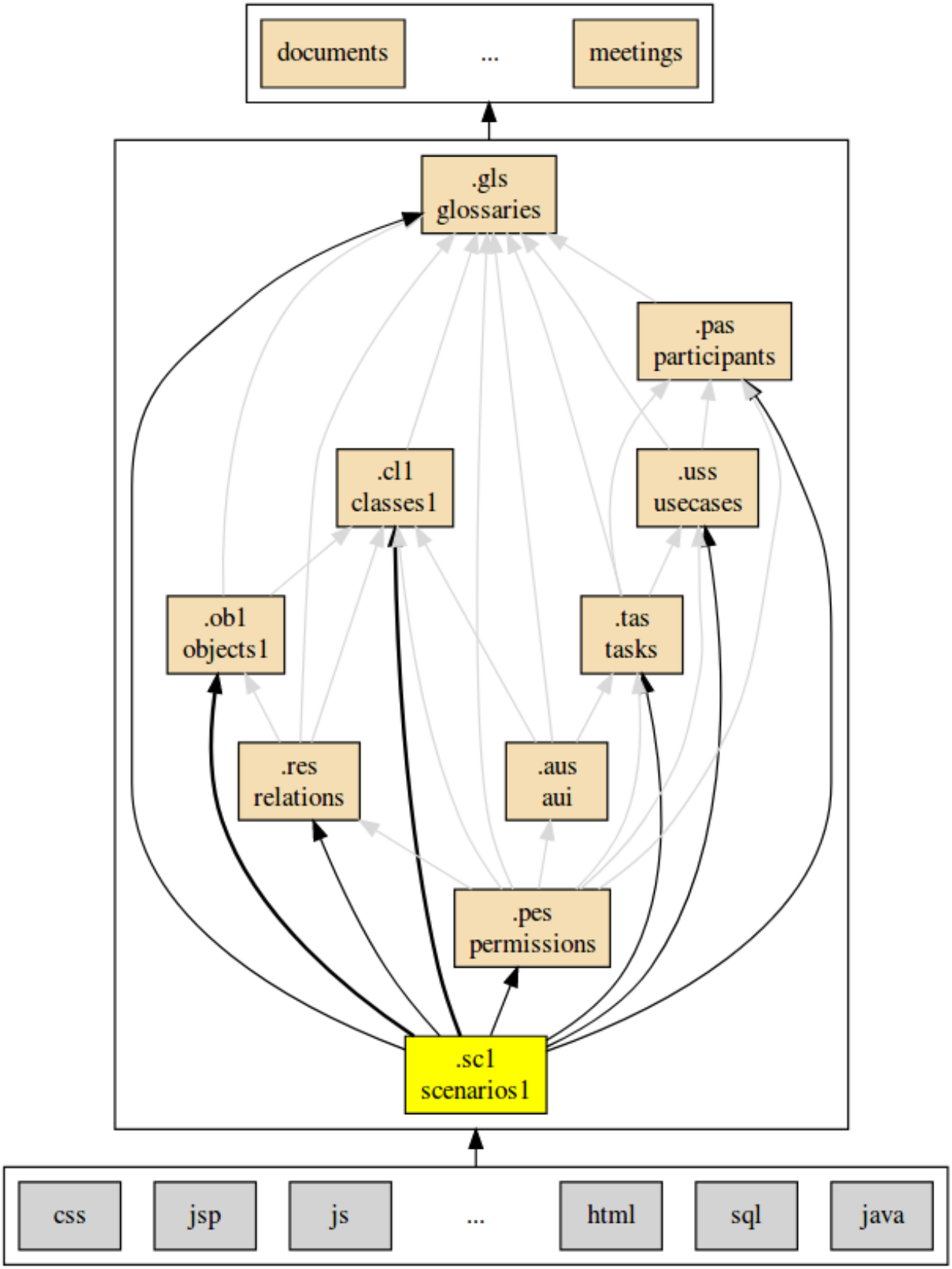
Concepts

Le langage ScenarioScript1 est basé sur les concepts suivants :

- les phrases
- les instructions
- les blocs de contexte
- les blocs de cas d'utilisation
- les scénarios textuels,
- les scénarios états,
- les scénarios cas d'utilisation

Dépendances

Le graphe ci-dessous décrit les dépendances entre langages.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

2.1.11 PermissionScript

Exemples

Version anglaise

```
permission model CyberCompagnie
import class model from '../classes/classes.cl1'
import usecase model from '../usecase/usecase.uss'

EmbaucherUnEmploye can create Employe, EstEmployeeDans
EmbaucherUnEmploye can create, update Compagnie.budget, Compagnie.quota
LicencierUnEmploye can delete Employe
Responsable, Secretaire can R Employee.salaire
Directeur can read, update Employee.salaire
```

Version française

```
permission model CyberCompagnie
import class model from '../classes/classes.cl1'
import usecase model from '../usecase/usecase.uss'

EmbaucherUnEmploye peut créer Employe, EstEmployeeDans
EmbaucherUnEmploye peut lire, modifier Compagnie.budget, Compagnie.quota
LicencierUnEmploye peut détruire Employe
Responsable, Secretaire peut L Employee.salary
Directeur peut lire, modifier Employee.salary
```

Note:

- `can` peut être remplacé par `peut`.
- Les actions peuvent être abbréviées ou pas (“C” ou “create”).
- Les actions peuvent être traduites. Voir la section *actions*.

Concepts

Conceptuellement le modèle de permissions est basé sur une suite de triplets :

```
(sujets, actions, ressources)
```

Ce triplet signifie : ” *les <sujets> peuvent effectuer les <actions> sur les <ressources>*.”

Exemple :

```
EmbaucherUnEmploye peut LM Compagnie.budget, Compagnie.quota
```

`EmbaucherUnEmploye` est le *sujet*. `L` et `M` sont les *actions*. `Compagnie.budget`, `Compagnie.quota` sont les *ressources*. Le triplet signifie : “*le cas d’utilisation* `EmbaucherUnEmploye` *peut lire et modifier les attributs* `budget` *et* `quota` *de la classe* `Compagnie`”.

Sujets

Un **sujet** est soit:

- un **acteur** (provenant du *modèle de participants*), par exemple `Directeur`,
- un **cas d'utilisation** (provenant du *modèle de cas d'utilisation*), par exemple `CreerUnDepartement`.

Actions

Les actions correspondent essentiellement au modèle CRUD (voir wikipedia). Les actions peuvent être écrites en entier ou sous forme abbréviées, en anglais ou en français.

En anglais	En français
C / create	C / créer
R / read	L / lire
U / update	M / modifier
D / delete	D / détruire
X / execute	X / exécuter

La signification des opérations dépend des ressources. Voir la section *ressources*.

Ressources

Pour un modèle de classe une **ressource** est soit :

- une **classe**, par exemple `Employe`,
- un **attribut**, par exemple `Employe.salaire`,
- une **opération**, par exemple `Employe.augmenter()`.
- une **association**, par exemple `EstAffecteA`,
- une **role**, par exemple `Employe.responsable`.

Le type de ressources définit les actions autorisées :

- l'opération **create/créer** s'applique à une classe ou à une association. Par exemple `créer Employe` ou `créer EstEmployePar`. Créer un attribut, un role ou une opération ne fait pas de sens.
- l'operation **read/lire** s'applique à un attribut ou à un role. Par exemple `lire Employe.salaire` ou `lire Employe.responsable`.
  - Lorsque cette action est associée à une classe (par exemple `lire Employe` alors n'importe quel attribut de la classe peut être lu (dans l'exemple l'accès est donné à tous les attributs de la classe `Employe`).
  - Lorsque cette action est associée à une association (par exemple `lire EstEmployePar`), alors celle-ci peut être traversée dans n'importe quel sens.
- l'opération **update/modifie** s'applique à un attribut (ou à une classe, de manière analogue à **read/lire**).
- l'opération **delete/détruire** s'applique à une classe ou à association
- l'opératop, **execute/exécuter** s'applique à une operation uniquement.

action/resc.	classe	attribut	operation	association	role
create	X			X	
read	[X]	X			X
update	[X]	X			
delete	X			X	
execute			X		

Méthode

Les tâches listées par la suite ne peuvent que difficilement être réalisées en séquentiel. Cependant plusieurs pratiques existent, selon que l'on part d'un modèle ou d'un autre.

Classes en premier

Dans la méthode "classes en premier" il s'agit de partir d'un modèle de classes, de lister chaque classes, attributs et associations, et dans chaque cas de répondre à la question *"qui change telle ou telle ressource ?"*. Le résultat pourrait être comme ci-dessous (résultats "triés" par la deuxième colonne) :

```
...  peut C Departement
...  peut L Departement.budget
...  peut M Departement.budget
...  peut D Departement
...
...  peut C Projet
```

Cette méthode permet de vérifier que toutes les parties du modèle de classes (à droite) sont utilisées "correctement".

Participants en premier

Considèrer le modèle de participants en premier revient à répondre à la question *"que peut faire tel ou tel acteur ?"* :

```
Directeur peut C ...
Directeur peut R ...
Directeur peut U ...
Directeur peut D ...
Secretaire peut C ...
Secretaire ...
...
```

Cette méthode permet de visualiser rapidemment les permissions associées à chaque acteur. Par contre le détail des cas d'utilisation est manquant.

Cas d'utilisation en premier

Partir des "cas d'utilisation en premier" revient à répondre à la question *"que peut faire tel ou tel cas d'utilisation ?"* :

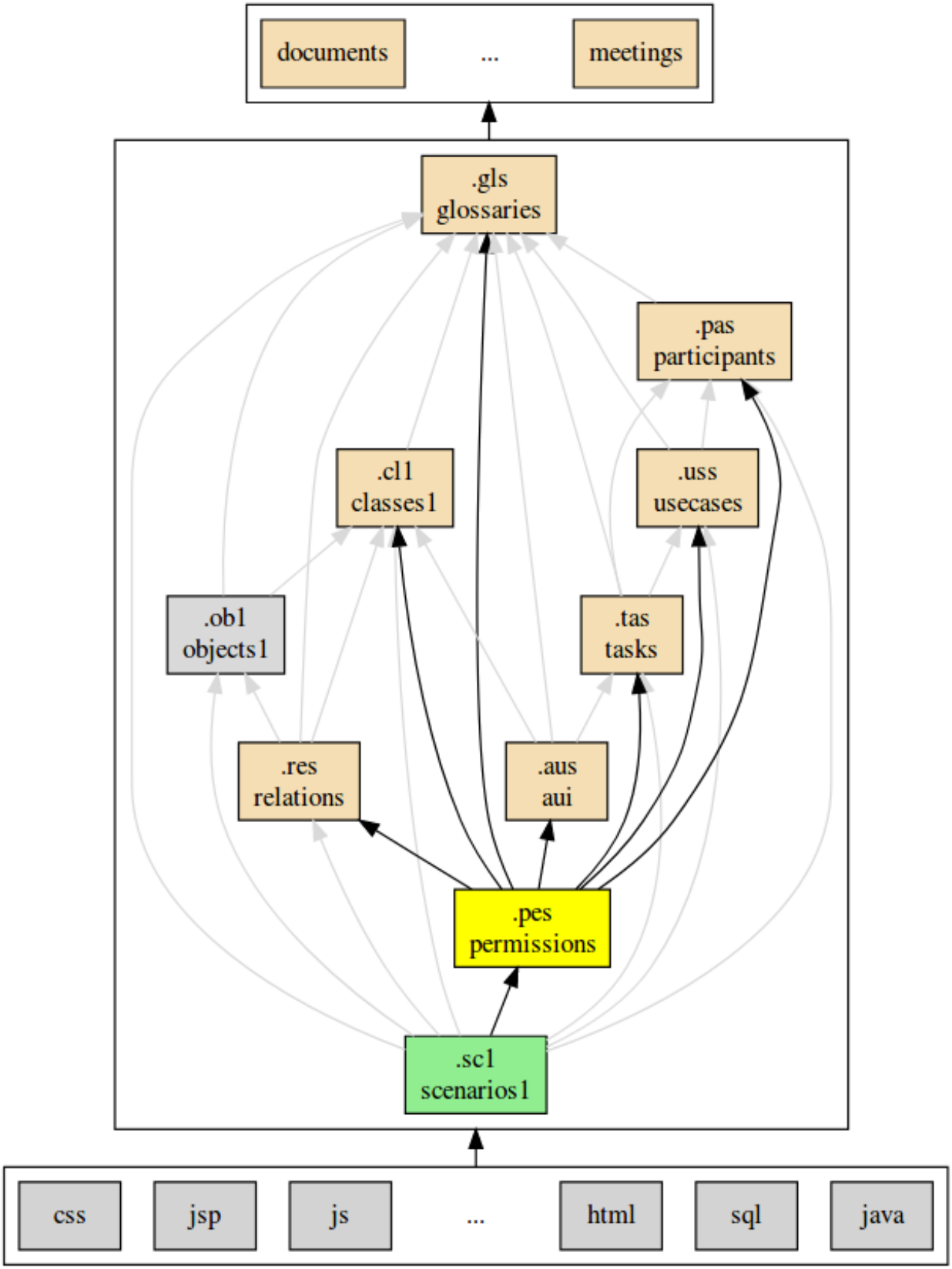
```
ReserverUneSalle peut C ...
ReserverUneSalle peut R ...
ReserverUneSalle peut U ...
ReserverUneSalle peut D ...
AugmenterUnEmploye peut C ...
AugmenterUnEmploye ...
...
```

Matrice

Les différentes techniques ci-dessus peuvent être combinées en produisant d'abord une matrice listant d'un coté toutes les resources (classes, etc.) et de l'autre tous les sujets (acteurs, etc.). Il s'agit ensuite de répondre pour chaque élément de la matrice à la question *"quelles actions peut être réalisées par ce sujet sur cette ressource"*.

Dépendances

Le graphe ci-dessous montre les dépendances entre langages.


documents
...
meetings
.gls
glossaries
.pas
participants
.cl1
classes1
.uss
usecases
.ob1
objects1
.tas
tasks
.res
relations
.aus
aui
.pes
permissions
.sc1
scenarios1
css
jsp
js
...
html
sql
java

CHAPTER 3

Artefacts

3.1 Artefacts

La méthode ModelScript défini une structure d'artefacts, c'est à dire de répertoires et de fichiers, avec des noms précis. Chaque tâche indique comment définir le contenu de chaque fichier.

3.1.1 concepts/

```
concepts/                   Description des concepts.

    besoins/                Résultats de la collecte/analyse des besoins
        ...

    glossaires/             Glossaires du projet.
        glossaires.gls      Glossaires exprimés en GlossaryScript.
        status.md           Status.

    classes/                Modèle de classes.
        classes.cl1         Modèle exprimé en ClassScript1.
        diagrammes/         Diagrammes de classes.
        status.md           Status.

    objets/                 Modéles d'objets.
        o<N>/               Modèle d'objets n°N (modèle positif)
            o<N>.ob1        Modèles d'objets exprimés en ObjectScript1.
            diagrammes/     Diagrammes d'objets en différents formats.
        ...                 ...
        on<N>               Modèle d'objets négatifs.
        ...                 ...
        status.md           Status.
```

3.1.2 cu/

```
cu/                         Description du comportement du système.

    cu/                     Modèle de cas d'utilisation.
        cu.uss              Cas d'utilisation en UsecaseScript.
        diagrammes/         Diagrammes de cas d'utilisation.
        status.md           Status.

    scenarios/              Scénarios conceptuels.
        s<N>/               Scénario n°N.
            s<N>.sc1        Scenario representé en ScenarioScript1.
            diagrammes/     Diagrammes liés au scénario.
        ...
        status.md           Status.

    permissions/            Modèle de permissions.
        permissions.pes     Modèle exprimé en PermissionScript.
        status.md           Status.
```

3.1.3 ihm/

```
ihm/                        Interface homme machine.

    taches/                 Modèles de tâches.
        <cu1>/              Modèle de tâches pour le cu <cu1>.
            taches-<cu1>.kxml      Modèle de tâches exprimé en KMade.
            taches-<cu1>.pdf       Modèle de tâches représenté en pdf.
        ...                 ...
        status.md           Status.

    ihm-abstraite/          Modèles d'interface abstraite.
        <cu1>/              Interface abstraite pour le cu <cu1>.
            ihma-<cu1>.pdf  Interface abstraite représentée en pdf.
        status.md           Status.

    ihm-concrete/           Interface concrète du système.
        charte-graphique.pdf    Charte graphique
        ...
        status.md           Status.

    evaluation/             Evaluation de l'interface homme machine.
        analyse/
            evaluation-heuristique.pdf
        tests/
            protocole.pdf
            rapport.pdf
        status.md           Status.
```

3.1.4 bd/

```
bd/                     Modèles et implémentaton de la base de données.
```

(continues on next page)

(continued from previous page)

```
    relations/          Modèle de relations.
        relations.res   Modèle de relations exprimé en RelationScript.
        status.md       Status.

    sql/                Implémentation SQL de la base de données.
        schema/         Schéma de la base de données.
            schema.sql  Schéma de la base de données exprimé en SQL.
        jdd/            Jeux de données.
            jdd<N>.sql  Jeux de données positif numéro N.
            jddn<M>.sql Jeux de données négatif numéro M.
            ...
        requetes/       Requêtes
            attendu/    Résultats attendus des requêtes
    cree-la-bd.sh       Script de création de la base de données.
    eval.sh             Script d'évaluation des requêtes
    status.md           Status.
```

3.1.5 projet/

```
projet/                 Informations liées au projet.
    aq                  Assurance qualité.
    sprint<N>/          Information à propos du Nième sprint.
        plannings/      Plannings pour le Nième sprint.
            previsionnel/
                planning-previsionnel.gan
                planning-previsionnel.gan.png
                planning-previsionnel.res.png
                planning-previsionnel.github.png
            intermediaire/
                planning-intermediaire.gan
                planning-intermediaire.gan.png
                planning-intermediaire.res.png
                planning-intermediaire.github.png
            effectif/
                planning-effectif.gan
                planning-effectif.gan.png
                planning-effectif.res.png
                planning-effectif.github.png
        audit/
            audit.pdf
            resume.md
        retrospective/
            retrospective.md
    suivi-du-temps/
        <XXX>.md
    suivis/
        suivis.trs
    done.md
    status.md
```

3.1.6 dev/

```
dev/                    Development artefacts including code.
    <CASESTUDY>/        Code containing the software
        status.md       Development status



participants/           Participant model.
    participants.pas    Participant model expressed in ParticipantScript.
    status.md           Work status.


playground/             Space for learning, prototyping, ...

status.md               Global status of the work.
```

CHAPTER 4

Tasks

4.1 Tasks

4.1.1 tâches concepts.*

tâche concepts.glossaires

résumé L'objectif de cette tâche est (1) de compléter le glossaire, (2) éventuellement de réécrire certains textes, en particulier ceux liés aux modèles d'objets et aux scénarios.

langage *GlossaryScript*

artefacts

- `glossaries/glossary.gls`
- `glossaries/status.md`

(A) Glossaire

Compléter le glossaire en respectant les règles du langage *GlossaryScript* ainsi que les *règles associées aux glossaires*.

(B) Textes

Le glossaire doit être aligné avec les termes utilisés par le client. En particulier si différents textes ont été fournis, ceux-ci devraient être réécrits. Pour la réécriture suivre les règles associées à la *réécriture de textes*.

D'un point de vue pragmatique si tous les textes ne peuvent pas être réécrits se focaliser en priorité sur la réécriture des textes associés aux modèles d'objets et aux scénarios.

(C) Identificateurs

Tout au long du projet il sera nécessaire de s'assurer en permanence de l'alignement avec le glossaire de tous les modèles mais aussi du code (SQL, Java, etc.). Cela impliquera entre autre d'aligner non seulement les textes, mais aussi les identificateurs. Lire les règles associées à la *réécriture d'identificateurs*.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche concepts.scenarios.textuels

résumé L'objectif de cette tâche d'obtenir, à partir des résultats de la capture des besoins, des scénarios textuels éventuellement accompagnés de modèles d'objets.

langage *ScenarioScript1*, *ObjectScript1*

artefacts

- `objects/o<N>/o<N>.ob1`
- `scenarios/s<N>/s<N>.sc1`

Introduction

L'activité de capture des besoins donne lieu à différents textes accompagnés d'autres ressources (images par exemple). Ces documents se trouvent dans le dossier `besoins`. L'objectif de cette tâche est de :

- extraire de ces documents une liste de scénarios accompagnés éventuellement de modèles d'objets,
- de produire des scénarios textuels épurés et basé sur le glossaire.

Il peut être utile de compléter le jeu de scénarios ainsi extrait par d'autres scénarios qu'il conviendra de faire valider auprès du client.

(A) Objets

Les documents fournis peuvent faire référence à des exemples ou à des jeux de données particuliers. Dans ce cas définir des modèles d'objets sous forme de texte. Associer à chaque modèle un identificateur, par exemple "o3". Créer ainsi le fichier `objets/o3/o3.ob1` en respectant la syntaxe du langage *ObjectScript1*. Dans un premier dans les modèles d'objets seront uniquement représentés sous forme de documentation (utilisation de `|`). Les modèles d'objets seront "codés" par la suite dans la *tâche concepts.objets*.

Si aucun jeu de données n'est fourni il peut être nécessaire d'en "inventer" un (ou plusieurs). Cela permettera d'instancier des scénarios comme indiqué ci-dessous. Les modèles d'objets serviront également de base pour valider le modèle de classes. Ils serviront finalement comme jeux de données pour valider le schéma de la base de données.

(B) Scénarios

Des élements correspondant à des scénarios ou parties de scénarios peuvent être fournis. Identifier ou numéroter les scénarios si ce n'est pas déjà fait. Cela donnera lieu par exemple à des fichiers comme `scenarios/s3/s3.sc1`.

Si les scénarios ne sont pas "instanciés" les instancier. Par exemple la phrase "*Le client achète des places pour un spectacle*" sera instancié en "*Paul achète 3 places pour 15€ pour le spectacle L'homme invisible programmé le 19/02/2020 à 17h15*". La liste des spectacles disponibles devra peut être être "inventée" si elle n'est pas présente. Elle pourra être définie sous la forme d'un modèle d'objets (voir ci-dessus) qui servira de "contexte" à ce scénario. Le fait que "*Paul achète 3 places*" fait par contre partie du scénario puisqu'il s'agit d'une action donnant lieu à un changement d'état.

Les scénarios auxquels on s'intéresse ici sont des scénarios systèmes : on s'intéresse au système plutôt qu'aux raisons pour lequelles les utilisateurs prennent telles ou telles décisions. Par exemple la phrase "*Après avoir amener son fils à l'école . . .* " peut être simplifiée en se concentrant sur les interactions avec le système. Les informations concernant les motivations de l'utilisateurs seront par contre intéressantes pour la conception d'interface homme machine. Mais il n'est pas nécessaire qu'elles figurent dans les scénarios auxquels on s'intéresse ici.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche concepts.classes

résumé L'objectif de cette tâche est (1) de définir/compléter le modèle de classes, (2) de le compiler, (3) de vérifier l'alignement du modèle de classes avec le glossaire si celui-ci existe.

langage *ClassScript1*

artefacts

- `concepts/classes/classes.cl1`
- `concepts/classes/status.md`

(A) Classes

Créer le modèle de classes dans le fichier `concepts/classes/classes.cl1`. Pour écrire le modèle utiliser la documentation du langage *ClassScript1* (USE OCL).

Eventuellement un modèle est peut être fourni, soit sous forme textuelle dans le fichier `concepts/classes/classes.cl1`, soit sous forme de diagramme dans le répertoire `concepts/classes/diagrammes`. Dans les deux cas il s'agit de compléter et/ou de corriger le modèle à partir des éléments fournis.

Le modèle de classes doit *in fine* être développé en fonction :

- des besoins exprimés par le client (dossier `concepts/besoins/`)
- des modèles d'objets (dossier `concepts/objets/`)
- des scénarios (dossier `cu/scenarios/`)
- de vos connaissances du domaine.

Certains éléments peuvent ne pas être présents.

(B) Compilation

Le modèle de classes doit **IMPERATIVEMENT** pouvoir être "compilé" sans erreur en utilisant la commande suivante (à partir du répertoire racine) :

```
use -c concepts/classes/classes.cl1
```

S'il y a des erreurs elles seront affichées. Aucun affichage signifie que le modèle est conforme au langage *ClassScript1* (USE OCL).

(C) Glossaire

Si un glossaire existe, vérifier que les termes importants apparaissant dans les noms de classes, d'associations, d'attributs ou de rôles sont bien dans le glossaire. Par exemple il peut être important de définir ce qu'est la "DateDeRetour" dans le contexte d'un bibliothèque. Ce terme fait partie du domaine. Il est sans doute nécessaire de le définir s'il ne correspond pas à une définition de sens commun. D'ailleurs le terme à définir est peut être "Retour".

Lire et appliquer les *règles associées à la réécriture d'identificateurs*.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche concepts.classes.diag

résumé L'objectif de cette tâche est de créer un ou plusieurs diagrammes de classes.

langage *ClassScript1*

artefacts

- `concepts/classes/diagrammes/classes.cld.clt`
- `concepts/classes/diagrammes/classes.cld.png`
- `concepts/classes/diagrammes/<NOM>.cld.clt`
- `concepts/classes/diagrammes/<NOM>.cld.png`

(A) Diagramme global

Il s'agit tout d'abord de créer un "diagramme global" de classes. Un tel diagramme doit présenter toutes les classes du modèle ; par opposition avec les "vues" qui ne présentent que certaines classes sélectionnées.

Note: Si une ébauche de diagramme a été fournie le diagramme dessiné devra en respecter la disposition.

Pour créer un diagramme de classes utiliser la commande suivante :

```
use concepts/classes/classes.cl1
```

Se référer ensuite à la documentation de USE OCL sur ScribesTools.

Respecter impérativement les consignes suivantes :

- (1) Sauvegarder le diagramme dans le fichier `concepts/classes/diagrammes/classes.cld.clt` (remplacer le fichier existant). Utiliser pour cela la commande `Save Layout` comme indiqué dans la documentation. NOTE: Le fichier `.clt` ("CLass Layout") contient le diagramme, c'est à dire le "graphique", la disposition des classes et des associations dans l'espace. Par opposition le fichier `.cl1` contient le modèle de classes, sans aucun rendu graphique.
- (2) Faire ensuite une copie d'écran du diagramme et remplacer le fichier `concepts/classes/diagrammes/classes.cld.png` fourni. Respecter **impérativement** les noms de fichiers, entre autre l'extension `.png`.

Attention: les diagrammes doivent impérativement montrer les cardinalités ; afficher si possible les noms de rôles ou d'associations si le diagramme reste visible. Utiliser le menu contextuel (click droit) pour avoir accès à ces options.

(B) Vues

Il peut vous être demandé de réaliser plusieurs diagrammes de vues. Contrairement au diagramme global qui montre toutes les classes (et est parfois peu lisible), une "vue" ne montre que certaines classes sélectionnées, en montrant par exemple le détail de ces classes.

Les classes à masquer peuvent être définies avec le menu contextuel de l'outil (click droit)

Tout comme pour le diagramme global, les fichiers à produire sont `concepts/classes/diagrammes/<NOM>.cld.clt|png` où `<NOM>` fait référence au nom de la vue, `.clt` fait référence au diagramme et `.png` à la capture d'écran correspondante.

tâche concepts.objets

résumé L'objectif de cette tâche est (1) de traduire les modèles d'objets textuels en modèle d'objets annotés en *ObjectScript1* et (2) de compiler ces modèles d'objets.

langage *ObjectScript1*

artefacts

- `concepts/objets/o<N>/o<N>.ob1`
- `concepts/objets/status.md`

Introduction

Cette tâche consiste à traduire les modèles d'objets décrits sous forme textuelle en modèles d'objets annotés. Chaque modèle d'objets se concrétise en un fichier `.ob1`. Ces fichiers vont être utilisés pour valider le modèle de classes (fichier `.cl1`). Il s'agit de répéter les étapes ci-dessous pour chaque modèle d'objets dans le répertoire `concepts/objets/`.

(A) Traduction

Le fichier `concepts/objets/o<N>/o<N>.ob1` (où `o<N>` est l'identifiant du modèle d'objet) contient un modèle d'objets décrit en langue naturelle. Il s'agit de traduire chaque ligne en utilisant le langage *ObjectScript1*.

NOTE: les modèles d'objet annotés peuvent être "compilés". Voir la section suivante pour plus de détails.

Lorsqu'une valeur n'est pas définie utiliser une instruction `... := Undefined`. Dans certains cas il peut être pertinent "d'inventer" une valeur ou des valeurs. Dans ce cas mettre une note dans le modèle de suivi.

Faire au mieux sachant que l'objectif est de traduire un texte fourni par (ou écrit en collaboration avec) le "client". Il sera peut être nécessaire de voir avec lui comment compléter/valider la description d'un modèle d'objets sachant qu'un tel modèle pourra par la suite être utilisé pour établir des tests et en particulier des tests de recette.

(B) Classes

Lorsqu'un modèle d'objet est créé il est **IMPERATIF** de Vérifier qu'il est aligné avec le modèle de classes. Pour cela utiliser la commande suivante :

```
use -qv concepts/classes/classes.cl1 concepts/objets/o<N>/o<N>.ob1
```

ou <N> correspond au numéro du modèle d'objet. L'interpreteur affichera les éventuelles erreurs de syntaxe ainsi que les erreurs de types ou de cardinalités. Si rien ne s'affiche cela signifie qu'aucune erreur n'a été trouvée.

La localisation des erreurs n'est parfois pas indiquée clairement. Si ce problème apparaît utiliser l'interpreteur USE en utilisant la commande suivante :

```
use -nogui concepts/classes/classes.cl1 concepts/objets/o<N>/o<N>.ob1
```

Si l'objectif est de vérifier les cardinalités utiliser ensuite la commande use `check` ou `check -v`. Terminer finalement avec la command `quit` ou `Ctrl C` pour sortir de l'interpréteur.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche concepts.objets.diag

résumé L'objectif de cette tâche est de créer des diagrammes d'objets.

langage *ObjectScript1*

artefacts

- `concepts/objets/o<N>/diagrammes/o<N>.obd.olt`
- `concepts/objets/o<N>/diagrammes/o<N>.obd.png`

(A) Diagrammes

Pour produire un diagramme d'objets procéder de manière analogue à la production de diagrammes de classes.

Lancer l'outil USE avec la commande suivante (remplacer <N> par le numéro du modèle d'objets) :

```
use concepts/classes/classes.cl1 concepts/objets/o<N>/o<N>.ob1
```

Utiliser le menu `View > Create View > Object Diagram`.

De manière analogue à la création de diagramme de classes il s'agit de produire deux fichiers :

- le diagramme lui même, `concepts/objets/o<N>/diagrammes/o<N>.obd.olt` (".olt" est un acronyme pour OBject Layout).
- le diagramme sous forme de copie d'écran (`concepts/objets/o<N>/diagrammes/o<N>.obd.png`).

NOTE1: La disposition des objets doit autant que possible refléter la disposition du diagramme de classes.

NOTE2: La création des diagrammes d'objets et de classes est similaire dans le principe. Se référer à la *tâche concepts.classes.diag* si nécessaire).

(B) ObjetsIsoles

Observer la présence ou non d'objets isolés dans le diagramme, des objets ou groupe d'objets connectés à aucun autre. Vérifier s'il s'agit d'un problème dans le modèle d'objets textuel lui même ou un problème dans la traduction qui en a été faite.

(.. tâche concepts.objets.negatifs:

tâche concepts.objets.negatifs

résumé L'objectif de cette tâche est (1) de créer des modèles négatifs d'objets et (2) de vérifier que ces modèles génèrent les erreurs escomptées.

langage *ObjectScript1*

artefacts

- `objets/no<N>/no<N>.ob1`
- `objets/status.md`

Introduction

Tester une application requière la production de tests dits "positifs" et de tests dits "négatifs".

- Les tests positifs testent l'application dans les cas de fontionnement normaux. Aucune erreur ne doit être générée. Ce sont les tests "normaux".
- Les tests négatifs testent l'application dans les cas aux limites. Il s'agit de vérifier en situation d'erreurs que l'application détecte bien les erreurs escomptées.

Les modèles positifs d'objets (ou modèles d'objets positifs) peuvent être vus comme des tests positifs du modèle de classes. Ils montrent que le modèle d'objets est conforme au modèle de classes. Souvent le terme "positif" est omis et on parle de "modèles d'objets" par simplification, à la place de "modèles positifs d'objets".

Quant à eux les modèles négatifs d'objets (ou modèles d'objets négatifs) testent le modèle de classes aux limites. Ils ne sont pas conformes au modèle de classes. Des erreurs (violations) doivent être détectées.

Attention: Seules certains types de violations sont détectables automatiquement. Voir ci-dessous pour plus de détails.

(A) Création

Les modèles négatifs d'objets donnent lieu soit à des violations de cardinalités, soit à des violations de contraintes. Il s'agit dans cette tâche d'écrire un ou plusieurs modèles d'objets négatifs. Chaque violation attendue doit être déclarée dans le modèle. Se référer à la documentation concernant les *Violations* pour plus de détails.

En pratique il s'agit de créer des modèles d'objets négatifs ou de compléter ceux existants (le cas échéant). Ces modèles doivent être rangés dans des répertoires `on<N>` (où `<N>` est un numéro). La structure de ce répertoire doit être analogue à la structure des modèles d'objets.

Pour définir le contenu de ces modèles, choisir d'abord une liste de violations intéressantes à détecter. Définir ensuite le contenu du modèle d'objets pour que ces erreurs soient produites.

Il est possible de créer de multiples modèles d'objets, un modèle par violation, ou au contraire de rassembler en quelques modèles d'objets toutes les violations. Indiquer dans le fichier `concepts/objets/status.md` les choix retenus pour structurer les modèles d'objets.

(B) Vérification

Une fois les violations déclarées dans les modèles d'objets négatifs il s'agit de vérifier, pour chaque modèle, que les violations ont bien lieu. Utiliser pour cela l'outil USE. Les violations de cardinalités sont indiquées dans la section `Checking structure...`. Les violations de contraintes sont indiquées dans la section `Checking invariants...`. Si une violation n'est pas détectée, soit la contrainte (ou la cardinalité) est erronée (ou non implémentée en OCL), soit le modèle d'objets est incorrect.

> **Attention:** Les violations de contraintes ne sont détectées que pour les contraintes exprimées en OCL. Les contraintes exprimées uniquement sous forme textuelle sont ignorées par l'outil USE.

En pratique il s'agit d'observer les messages d'erreurs produits par l'outil USE et de comparer ces erreurs aux instructions `violates` introduites dans les modèles négatifs d'objets. Consigner le résultat dans le fichier `status.md`.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.)

tâche concepts.contraintes.ln

résumé L'objectif de cette tâche est d'exprimer en langage naturel les contraintes devant être ajoutées au modèle de classes.

langage *ClassScript1*

artefacts

- `classes/classes.cl1`

(A) Contraintes

Un modèle de classes doit comporter non seulement des classes et des associations mais aussi les contraintes qui leur sont associées.

Note: Il existe trois types de contraintes : (1) les invariants, (2) les préconditions et (3) les postconditions. Ces deux dernières catégories sont associées aux opérations. Sachant que l'on ne considère pas ces dernières au niveau conceptuel, nous nous limiterons ici au invariants, et par abus de langage le terme "contrainte" sera utilisé pour "invariant".

Dans cette tâche les invariants associés au modèle de classes doivent être décrits en Langue Naturelle (LN).

Par la suite ces invariants pourront être traduites en langage OCL, mais ce n'est pas l'objectif ici.

Pour réaliser cette tâche utiliser la *syntaxe des contraintes en langue naturelle*. Pour savoir comment "trouver" ces contraintes utiliser la *méthode de définition des contraintes*.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche concepts.contraintes.ocl

résumé L'objectif de cette tâche est de traduire les contraintes exprimées en langage naturel en contraintes OCL.

langage *ClassScript1*

artefacts

- `classes/classes.cl1`

(A) Contraintes

L'expression des contraintes en langage naturel est indispensable pour garantir l'alignement avec les contraintes métier (business rules). Sans cela le logiciel ne correspondera pas au besoin du client. Dans cette tâche il s'agit d'aller plus loin en formalisant ces contraintes en langage OCL, le langage standardisé d'UML pour les contraintes. Voir la feuille de résumé OCL pour des précisions sur le langage OCL. Vérifier que la traduction en OCL est fidèle à la contrainte en langage naturel.

Note: Pour rappel, par abus de langage nous utilisons les termes "contraintes" et "invariants" de manière interchangeable. Les autres types de contraintes (pré et post conditions) ne sont pas considérée dans la mesure où les opérations ne sont pas prises en compte.

(B) Tests positifs

Vérifier que l'ensemble des modèles d'objets (positifs) ne génèrent aucune erreur. Utiliser la commande `use -qv` pour cela. Si des erreurs sont produites cela veut dire que les contraintes sont trop restrictives.

(C) Tests négatifs

Vérifier que les tests négatifs concernant telle ou telle contrainte produisent bien les violations escomptées. Si toutes les violations attendues ne sont pas produites alors c'est que les contraintes écrites ne sont pas assez restrictives. Revoir les contraintes dans ce cas là.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche concepts.scenarios.etats

résumé L'objectif de cette tâche est (1) de traduire les scénarios textuels en scénarios états, (2) de valider la conformité de ces scénarios par rapport au modèle de classes.

langage *ScenarioScript1*

artefacts

- `cu/scenarios/s<N>/s<N>.sc1`
- `cu/scenarios/s<N>/diagrammes/s<N>.scd.olt`
- `cu/scenarios/s<N>/diagrammes/s<N>.scd.png`

Introduction

L'objectif de cette tâche est de traduire dans un premier temps les scénarios textes en scénarios "états", c'est en scénarios vu comme une simple succession de changements d'états effectuée via des d'instructions `!`. Le terme "scénarios états" doit être considéré en regard aux "sénarios cas d'utilisation". Ces derniers seront produits par la suite dans la *tâche cu.scenarios*.

Dans cette tâche on ne s'intéresse qu'au changement d'états du système en adoptant une perspective "données".

Les tâches ci-dessous doivent être répétées pour chaque scénario présent dans le répertoire `cu/scenarios/`.

(A) Traduction

En pratique, comme dans les modèles d'objets, il s'agit dans cette tâche simplement de traduire le texte des scénarios en une suite d'instructions `!` *à plat*. Voir la tâche (*tâche concepts.objets*) pour plus de détails.

Note: Si le fichier `s<N>.sc1` n'est pas vide ignorer les éventuelles instructions comme ci-dessous :

```
--@ context
    ...
--@ end
--@ ... va ...
    ...
--@ end
```

Ignorer également les emboîtements correspondants, s'ils sont présents.

(B) Classes

Vérifier que le scenario est aligné avec le modèle de classes. Pour cela utiliser la commande suivante à partir du répertoire principal

```
use -qv concepts/classes/classes.cl1 cu/scenarios/s<N>/s<N>.sc1
```

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

4.1.2 tâches cu.*

tâche cu.participants

résumé L'objectif de cette tâche est de définir les acteurs et les personnages.

langage *ParticipantScript*

artefacts

- `participants/participants.pas`

(A) Acteurs

Définir tout d'abord les "acteurs" qui interviendront par la suite dans le modèle de cas d'utilisation. Donner pour chaque acteur un nom (p.e. `ResponsableDesAchats`) ainsi qu'une courte définition faisant référence aux éléments du glossaire. Par définition un acteur intéragit directement avec le système (via une interface), sinon il ne s'agit pas d'un acteur.

(B) Personnages

Les "personnsages" sont des instances particulières d'acteurs. Ceux-ci interviennent dans les modèles de scénarios. Repérer "qui", dans chaque scénario, existant ou à définir, joue le rôle d'un acteur. Définir chaque personnage en donnant a minima son nom et son type. Par exemple `mario : ResponsableDesAchats`.

(C) Alignement

Les participants et les personnages pourront être définis "à la demande" lors de la *tâche cu.preliminaire* ou de la *tâche cu.scenarios*. Par la suite il s'agira aussi d'aligner les participants au modèle de permissions.

In fine le modèle de particiants doit être aligné avec les modèles suivants:

- le modèle de cas d'utilisation
- les différents modèles de scénarios.
- le modèle de permission.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche cu.preliminaire

résumé L'objectif de cette tâche est de définir un modèle préliminaire de cas d'utilisation, à savoir la liste des acteurs, des cas d'utilisation et des interactions.

langage *UsecaseScript*, *ParticipantScript*

artefacts

- `participants/particiants.pas`
- `usecases/usecases.uss`

Introduction

Il s'agit dans cette tâche de définir un **modèle préliminaire** et en particulier d'être très synthétique lors des différentes descriptions. Cette approche est classique dans un démarche agile. On ne cherche pas à définir de manière détaillée chaque élément mais au contraire à donner une vision globale du système. Une fois les priorités établies pour chaque cas d'utilisation, ces cas d'utilisation seront définis peu à peu de manière bien plus précise dans *tâche cu.detail*.

(A) Acteurs

Les acteurs doivent être définis de manière séparée dans le *modèle de participants* via la tâche *tâche cu.participants*.

(B) Interactions

Définir quel(s) acteur(s) peut réaliser tel ou tel cas d'utilisation. On définira dans un premier temps les cas d'utilisation juste par leur nom. Ils seront décrits dans l'étape suivante.

(C) Cas d'utilisation

Les cas d'utilsation doivent être accompagnés d'une très brève description résumant

- (1) les objectifs de l'acteur et
- (2) le déroulement "normal" du cas d'utilisation.

La description doit être généralisée. Contrairement au scénarios un cas d'utilisation reflête le cas général : écrire `Le client s'identifie` plutôt que `Paul tape son code 6535`.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche cu.detail

résumé L'objectif de ce modèle est de décrire de manière détaillée un ou plusieurs cas d'utilisation en les sélectionnant parmi les cas d'utilisation du modèle préliminaire.

langage *UsecaseScript*

artefacts

- `usecases/usecases.uss`

(A) Définition

Une fois le modèle préliminaire défini via la *tâche cu.preliminaire*, décrire de manière plus détaillée le où les cas d'utilisation les plus prioritaires. Se reporter à la documentation de *UsecaseScript* pour voir quels aspects peuvent/doivent être décrits.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche cu.scenarios

résumé L'objectif de cette tâche est d'aligner les scénarios définis jusque là avec les cas d'utilisation.

langage *ScenarioScript1*, *ParticipantScript*

artefacts

- `participants/participants.pas`
- `scenarios/n/scenario.scs`

Introduction

Les scénarios réalisés jusque là étaient caractérisés par le modèle d'objets qu'ils généraient à la fin de leur exécution. Il s'agissait d'une liste "à plat" de créations d'objets et de liens. Dans cette tâche les scénarios sont considérés comme un emboîtement d'instances de cas d'utilisation.

Les modèles de scénarios à raffiner/compléter se trouvent dans les fichiers `scenarios/s<N>/s<N>.sc1` (où `<N>` est un entier). Se reporter à la documentation de *ScenarioScript1* lorsque nécessaire.

(A) Personnages

Dans un premier temps il s'agit de repérer dans les scénarios quels "personnages" interagissent **directement** avec le système. Ces personnages sont, par définition, des instances d'acteurs (ajouter les acteurs si nécessaire).

A chaque fois qu'un personnage est identifié celui-ci doit être ajouté au modèle de participants (fichier `participants/participants.pas`).

Par exemple le personnage `marie` peut jouer le rôle de `Bibliothecaire` dans un scénario :

```
persona marie : Bibliothecaire
```

Les personnages comme “marie” peuvent être caractérisés par de nombreuses propriétés (voir l’exemple dans la documentation de *ParticipantScript*). Ces propriétés seront nécessaires par exemple lors de la conception d’Interfaces Homme Machine (IHM). Ici on se contentera d’une brève description pour chaque acteur.

(B) Décomposition

Chaque scénario état doit ensuite être décomposé sous forme d’une série d’instances de cas d’utilisation. Autrement dit il s’agit de repérer dans le texte des scénarios quels cas d’utilisation sont mis en oeuvre.

Soit le fragment ci-dessous :

```
...

--@ marie va RentrerUneOeuvre
    --| Stéphanie rend le disque "High way to hell" [A24].
    --| Marie scanne le disque qui devient à nouveau disponible [A26].
        ! insert (b4885, bib14) into EstDisponibleA
        ! delete (steph, emp1) from AEmprunte

...
```

Dans cet exemple le personnage `marie` intéragit (mot-clé `va`) avec le cas d’utilisation `RentrerUneOeuvre`. Cette interaction modifie l’état du système via les instructions `! insert ...` et `! delete ...`. L’indentation montre que les phrases/instructions emboitées font partie de cette instance de cas d’utilisation.

Les règles suivantes doivent être respectées :

- le cas d’utilisation `RentrerUneOeuvre` doit être défini dans le modèle de cas d’utilisation,
- `marie` doit être un personnage existant dans le modèle de participant et
- `marie` doit correspondre à un acteur (`Bibliothecaire` ici) pouvant exécuter le cas d’utilisation (une `Bibliothecaire` peut effectivement `RentrerUneOeuvre`).

(C) Contexte

Dans cette sous-tâche il s’agit d’isoler les instructions qui font partie du “contexte” plutôt que du flôt normal du scénario. Considéront par exemple le cas d’une réservation de salle. L’instruction `create s203 : Salle` ne fait pas partie du cas d’utilisation `ReserverUneSalle` car la salle `s203` pré-existe à l’exécution du cas d’utilisation : la salle n’est pas créée, elle est juste réservée ! Le fait que la salle 203 existe fait partie du “contexte”. Une telle information (la phrase et les instructions correspondantes) doivent être inclues dans un block `context`. L’exemple ci-dessous montre la séparation entre un block contextuel et un block de cas d’utilisation.

```
--@ context
    --| La salle 203 peut accueillir 30 personnes [A6].
        ! create s203 : Salle
        ! s203.capacite := 30
    --| Elle se trouve dans le batiment C [A7]
        ! create batC : Batiment
        ! insert (s203, batC) into EstDansBatiment

...
```

(continues on next page)

(continued from previous page)

```
--@ toufik va ReserverUneSalle
    --| Toufik décide de réserver la salle 203 [A21][A22].
        ! insert (toufif, s203) AReservee
        ! s203.reservations := s203.reservations + 1
    --| Il indique que l'évenement qu'il prépare est payant [A23].
        ...

...
```

Il s'agit de :

- déplacer ces blocks en début de scénario et
- vérifier que cela ne provoque aucune erreur dans la "compilation" du scénario.

Note: Certaines phrases doivent dans certains cas être "coupées en deux" ou rephrasées, par exemple si un morceau d'une phrase existante fait partie du contexte et d'autres éléments d'un cas d'utilisation.

(D) Texte

Le texte fourni initialement et qui a donné lieu au scénario état doit, dans certains cas, être remanié. Par exemple de déplacement de blocks contextuels en début de scénario peut impliquer un remaniement de certaines phrases. Il en est de même lorsque les limites des scénarios sont établies.

Quelque en soit la raison, certaines phrases peuvent être déplacées, découpées, ou même supprimées.

Il n'y a pas de règle pour le remaniement du texte. L'équipe de développement, mais aussi le client, doivent cepandant pouvoir "lire" et utiliser le scénario tout au long du son cycle de vie. Une attention particulière devra être portée aux élements de traçabilité (e.g. `[A12][A14-A19]`).

(E) Transformation

L'exemple ci-dessous résume le processus global :

- (1) définition des personnages (`persona x : A`),
- (2) identification des instances de cas d'utilisation (`x va y`),
- (3) extraction des instructions du contexte (`context`),
- (4) remaniement du texte.

```
============================ ========================================
  AVANT: Scénario état          APRES: Scénario cas d'utilisation
============================ ========================================

                             Modele de participant (participant.pas)
                             ---------------------------------------
                                     participant marie : Bibliotecaire
                                     participant toufik : Manager

                                     ...

                             Modèle de scenario (S<N>.sc1)
```

(continues on next page)

(continued from previous page)

```
                            ----------------------------------------

--| phrase1                 --@ context
--| phrase2                     --| phrase3 modifiée
    ! instruction1                  ! instruction3
    ! instruction2                  ! instruction4
--| phrase3
    ! instruction3          --@ toufik va ReserverUneSalle
    ! instruction4              --| phrase1
--| phrase4                     --| phrase2
--| phrase5                         ! instruction1
--| phrase6                         ! instruction2
    ! instruction5
    ! instruction6          --| phrase4 modifiée
    ! instruction7          --| phrase5
--| phrase7
    ! instruction8          --@ marie va RentrerUneOeuvre
--| phrase8                     --| phrase6
                                    ! instruction5
                                    ! instruction6
                                    ! instruction7
                                --| phrase7
                                    ! instruction8

                            --| phrase8

=========================== ========================================
```

(F) Cas d'utilisation

Vérifier (manuellement) que le modèle de scénarios est bien aligné avec le modèle de cas d'utilisation. Par exemple `toufik va ReserverUneSalle` implique que `toufik` est un personnage, qu'il est peut être `ChefBibliothequaire` et qu'un `ChefBibliothequaire` peut `ReserverUneSalle`.

Note: A l'heure actuelle cette vérification n'est pas outillée et elle doit donc être faite manuellement.

(G) Classes

Vérifier que le scénario est encore aligné avec le modèle de classes.

```
use -qv Classes/classes.cls Scenarios/n/scenario.scn
```

Cette vérification a été faite précédemment avec le scénario état mais il s'agit là de vérifier que la transformation ci-dessus n'a pas généré de problèmes supplémentaires. Ce peut être le cas si le réordonnancement des instructions n'est pas correct.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche cu.permissions

résumé L'objectif de cette tâche est de définir le modèle de permissions en mettant en relation les acteurs/cas d'utilisation avec les éléments du modèle de classes utilisés.

langage *PermissionScript*

artefacts

- `permissions/permissions.pes`

Introduction

Après avoir défini un modèle de classes (*tâche concepts.classes*) et un modèle de cas d'utilisation (*tâche cu.preliminaire*), il est possible de définir le modèle de permissions. Cette tâche a *in fine* pour objectif d'aligner :

- le modèle de permissions,
- le modèle de classes,
- le modèle de participants,
- le modèle de cas d'utilisation,
- les modèles de scénarios.

Il existe *plusieurs manières* de remplir le modèle de permission. L'objectif de cette tâche est de mettre en pratique deux de ces techniques.

(A) Technique 1

Commencer par la méthode *"classes en premier"*. Lorsque des classes/attributs/associations ne sont créés/utilisés/modifiés par aucun cas d'utilisation, indiquer pourquoi sous forme de commentaires dans le modèle de permissions. Ajuster le modèle de cas d'utilisation si nécessaire.

(B) Technique 2

Dans un deuxième temps utiliser la méthode *"Cas d'utilisation en premier"* pour remplir la suite du modèle. Ajuster le modèle de classes si nécessaire.

(C) Scénarios

Une fois le modèle de permission créé, vérifier que les accès réalisés dans les scénarios ne violent pas les permissions données.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

4.1.3 tâches ihm.*

tâche ihm.taches

résumé L'objectif de cette tâche est d'élaborer le modèle de tâches pour chaque cas d'utilisation prioritaires.

artefacts

- `ui-tasks/<CU>/<CU>.kxml`
- `ui-tasks/<CU>/<CU>.pdf`

(A) Tâches

Ce travail a pour but de créer le modèle de tâches en utilisant l'environment KMade. Pour chaque cas d'utilisation prioritaire `<CU>` devront être livrés : * le fichier `ui-tasks/<CU>/<CU>.kxml` * un fichier `ui-tasks/<CU>/<CU>.pdf`.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche ihm.abstraite

résumé L'objectif de cette tâche est définir les IHM abstraite.

langage KMade

artefacts

- `ui-abstract/<MdT>/<MdT>.pdf`

(A) IHM abstraites

Ce travail consiste à définir les interfaces abstraites correspondant à chaque modèle de tâches (MdT). Pour chaque modèle un document `ui-abstract/<MdT>/<MdT>.pdf` doit être produit. Il peut s'agir d'une copie d'écran, d'une photographie ou d'un diagramme réalisé avec un outil. Si d'autres fichiers sont produits ils porteront des noms tel que `ui-abstract/<MdT>/<MdT>.<ZZZ>`.

(B) Alignement

S'assurer que les modèles d'IHM abstraites sont alignés avec les modèles de tâches et correspondent à des transformations définies.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche ihm.concrete

résumé Cette tâche consiste à donner des éléments de définition pour l'interface concrète. Il s'agit ici de définir (1) la charte graphique et (2) éventuellement des maquettes de l'interface concrète.

artefacts

- `ui-concrete/*`

(A) Charte graphique

Elaborer puis livrer la charte graphique sous forme d'un fichier `ui-concrete/charte-graphique.pdf`.

(B) Maquettes

Elaborer les maquettes sous la forme de fichiers pdf dans le répertoire `ui-concrete/`.

(C) Alignement

S'assurer que les maquettes sont bien alignées avec les modèles de tâches.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche ihm.evaluation

résumé L'objectif de cette tâche est d'évaluer l'IHM produite.

artefacts

- `ui-evaluation/analysis/evaluation-heuristique.pdf`
- `ui-evaluation/tests/protocole.pdf`
- `ui-evaluation/tests/rapport.pdf`

(A) Evaluation experte

Le résultat de l'évaluation experte de l'interface doit être livré sous forme pdf dans `ui-evaluation/analysis/evaluation-experte.pdf`.

(B) Protocole

Avant de réaliser des tests utilisateurs le protocole de tests doit être élaboré puis livré sous forme pdf dans `ui-evaluation/tests/protocole.pdf`.

(C) Tests

Après avoir défini le protocole de tests utilisateurs, les tests peuvent avoir lieu. Une fois ces tests effectués le bilan doit être élaboré et livré dans le document `ui-evaluation/tests/rapport.pdf`.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

4.1.4 tâches bd.*

tâche bd.classes

résumé L'objectif de cette tâche est d'annoter le "modèle de classes conceptuel" afin de le transformer en un "modèle de données" pour base de données.

langage *ClassScript1*

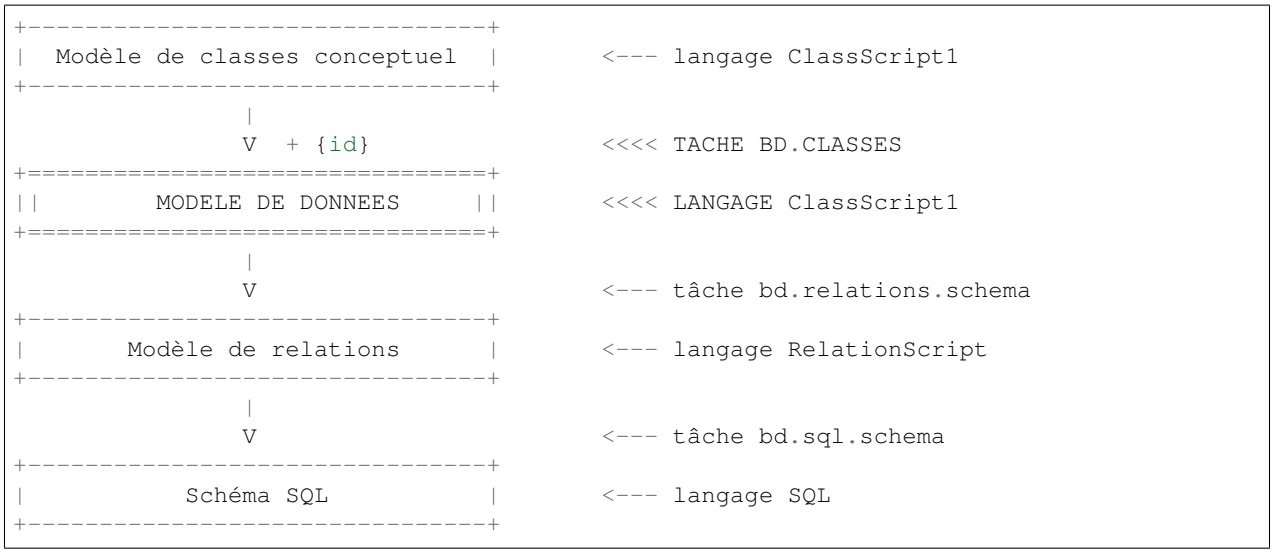
artefacts

- `concepts/classes/classes.cl1`

Introduction

Le modèle de classes élaboré jusqu'à présent était un modèle conceptuel, c'est à dire un modèle décrivant des concepts du domaine de manière abstraite ; et ce indépendamment de toute considération technique.

Il s'agit maintenant de transformer ce modèle abstrait en un "modèle de données" pour base de données (voir l'illustration ci-dessous). Ce modèle de données est un genre de modèle de classes. Il est donc écrit en *ClassScript1* comme tout modèle de classes. Sa particularité est qu'il contient des annotations `{id}` pour spécifier les clés.



Dans cette tâche il s'agit de préparer le modèle de classes conceptuel avant de le transformer en modèle de relations qui sera par la suite transformé en schéma SQL (cela fait l'objet de la *tâche bd.relations.schema* puis de la *tâche bd.sql.schema*).

Pour simplifier le modèle de données sera défini en lieu et place du modèle de classes conceptuel. Autrement dit dans cette tâche il est demandé de modifier le fichier `concepts/classes/classes.cl1`.

(A) Identifiants

Pour chaque classe, il s'agit de définir quels attributs ou quelles les combinaisons d'attributs forment une clé. En UML cette information prend généralement la forme d'annotations `{id}`.

Note: Rappelons que la notion de "clé" est propre au modèle relationnel. Dans le monde objet cette notion n'est normallement pas utilisée. Il n'y a pas besoin de "clés" car tout objet est systématiquement identifié de manière unique. Les annotations *{id}* sont donc uniquement utilisées dans le modèle de données en vue de la transformation vers le modèle relationnel.

Les annotations `{...}` n'étant pas disponibles en ClassScript1, on utilisera le suffixe `_id` pour les identificateurs clés. Par exemple l'attribut `login` devient `login_id`. Cette convention n'est pas parfaite mais elle permet de visualiser les clés dans les diagrammes de classes avec l'outil USE OCL.

(B) Identifiants multiples

Dans le cas de plusieurs clés candidates le suffixe sera numéroté ; par exemple `prenom_id1`, `nom_id1`, `numen_id2`. Voir le langage *RelationScript* pour d'autres exemples.

Une clé peut également être simplement suffixée par un simple caractère souligné _. De manière consistante avec le langage *RelationScript* ce suffixe _ signifie juste que l'attribut fait partie d'un identifiant, mais la notation ne spécifie pas lequel. Cette notation peut être choisie si l'on désire avant tout améliorer la lisibilité du diagramme. Lorsque la notation simplifiée _ est utilisée il n'y a pas d'ambiguité dans le cas d'un seul identifiant (par exemple `login_` seul). Par contre dans le cas de `prenom_`, `nom_`, `numen_` il n'est pas possible de déterminer qu'il y a deux clés. Par défaut et sans indication contraire on supposera qu'il existe une seule clé composée de tous les attributs "soulignés". Si ce comportement par défaut n'est pas adapté le détail des clés peut être indiqué sous forme de contraintes explicites. Utiliser pour cela la notation pour les contraintes textuelles (voir le langage *ClassScript1*).

Voir la *tâche bd.relations.schema* pour plus d'information sur la manière de spécifier les clés en *RelationScript*.

(C) Compositions

Un objet composant est parfois identifié par rapport à l'objet qui le contient. Par exemple dans un batiment une salle peut être identifiée en partie par son numéro, par exemple 127, mais aussi le nom du batiment, par exemple "condillac". Dans cet exemple l'identifiant de la salle est le couple ("condillac" , 127).

```
class Batiment
    attributes
        nom_id : String             -- exemple: "condillac"
end

composition Contient
    between
        Batiment[1] role batiment   -- composite : un Batiment
        Salle[*] role salles        -- composants : les Salles
end

class Salle                         -- clé : (nom_id,numero_id)
```

(continues on next page)

(continued from previous page)

```
    attributes
        numero_id : Integer          -- exemple 127
end
```

Le fonctionnement ci-dessus, l' "importation" de l'identifiant du composite, se fait dans le cadre d'une composition.

(D) Composition artificielles

Dans l'exemple ci-dessus la nature de l'association, une composition, est tout à fait logique. Un batiment est bien composé de salles. Par contre, pour les besoins de la transformations en base de données, il peut parfois être nécessaire de changer une association "standard" en une composition alors que cela n'est pas naturel. On parlera alors de "composition artificielle".

Par exemple :

```
association ComporteSeance
    between
        Salle[1] role salle
        Seance[*] role seances
end
```

peut être changé en une composition artificielle :

```
composition ComporteSeance
    between
        Salle[1] role salle
        Seance[*] role seances
end
```

Même si cette composition pourrait sembler contestable dans le cas d'un modèle conceptuel, cette modification peut être valide dans un modèle technique, ici dans le cadre de la conception de bases de données.

(E) Classes associatives

Selon le standard UML l'identifiant d'une classe associative est formé des identifiants des deux classes de chaque coté de la classe associative. Considérons la classe associative suivante :

```
class Personne
    attributes
        nom_id : String
end

class Societe
    attributes
        siren_id : String
end

associationclass Emploi
    attributes
        salaire : Integer
    between
        Personne[*] role employes
        Societe[*] role employeurs
end
```

Le standard UML indique explicitement que la clé de la classe `Emploi` est (`nom_id`, `siren_id`).

En complétant cet exemple un emploi pourrait de plus être identifié par un attribut clé `nnue_id` (nnue signifiant par exemple Numéro National Unique d'Emploi). Dans ce cas `nnue_id` est une autre clé candidate.

(F) Classes associatives artificielles

Dans la modélisation précédante on ne modélise que l'état des employés à un moment donné. La sémantique du standard d'UML indique en effet *"il n'y a qu'un emploi entre une personne et une société donnée"*.

Ainsi on ne peut donc pas modéliser le fait que "paul" a travaillé la première fois en 2007 à dans à la société "Mega-Tron" et une deuxième fois en 2020. Dans cette situation il y a deux emplois entre la même société et la même personne. Situation impossible à modéliser avec le modèle ci-dessus. Pour parlier ce problème on introduit la notion de "classe associative artificielle".

Supposons en effet que l'on veuille modéliser l'historique des emplois. Une personne (par exemple paul) peut donc avoir tenu plusieurs emplois dans la même société mais en débutant à des années différentes (pour simplifier on consière uniquement la granularité des années dans cet exemple). La classe associative est modifiée comme suit :

```
associationclass Emploi
    attributes
        salaire : Integer
        nnue_id : String
        annee_lid : Integer
    between
        Personne[*] role employes
        Societe[*] role employeurs
end
```

Comme on peut le voir l'attribut `annee` a été suffixé avec le suffixe `_lid` ("lid" pour "local id").

Dans cet exemple il y a deux clés candidates pour la classe `Emploi` :

- (`nnue_id`)
- et (`nom_id`, `siren_id`, `annee_lid`).

Le numéro national unique d'emploi (nnue) est une clé "globale" associée à la classe associative `Emploi` (comme elle l'aurait été à n'importe qu'elle autre classe, une clé associative étant une classe).

La clé (`nom_id`, `siren_id`, `annee_lid`) est liée au fait que `Emploi` est une classe associative artificielle. En pratique l'attribut `annee_lid` (local id) a été ajouté aux deux clés "importées" des deux classes de "chaque coté".

> **Attention:** L'utilisation de classe associative artificielle, c'est à dire du préfixe `_lid` est complètement incompatible avec le standard UML. Cette convention est pratique dans le cadre du développement de modèles de données en vue de transformation vers le modèle relationnel, mais attention à ne pas utiliser cette convention hors de ce contexte !

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche bd.relations.schema

résumé Cette tâche a pour objectif de créer le schéma relationnel à partir du modèle de données (si celui-ci existe).

langage *RelationScript*

artefacts

- `bd/relations/relations.res`

Introduction

Le schéma de production d'une base de données à partir d'un modèle de classes est le suivant.

```
+---------------------------------+
|  Modèle de classes conceptuel   |       <--- langage ClassScript1
+---------------------------------+
                 |
                 V                        <--- tâche bd.classes
+---------------------------------+
|         Modèle de données       |       <--- langage ClassScript1
+---------------------------------+
                 |
                 V                        <--- TACHE BD.RELATIONS.SCHEMA
+=================================+
||       MODELE DE RELATIONS     ||       <--- LANGAGE RelationScript
+=================================+
                 |
                 V                        <--- tâche bd.sql.schema
+---------------------------------+
|            Schéma SQL           |       <--- langage SQL
+---------------------------------+
```

Dans un tel contexte on s'intéresse à la troisème étape. Sinon, il s'agit simplement de créer un modèle de relation à partir de zéro.

Note: Dans cette tâche seul le schéma de données est considéré. On ne prend pas en compte d'éventuels jeux de données (datasets).

Le fichier a modifier dans cette tâche est `bd/relations/relations.res`. *RelationScript* est le langage utilisé. Se référer à la documentation pour plus d'exemples.

(A) Columns

La première étape consiste à définir les relations et leurs colonnes.

```
relation LesAppartements
    columns
        nom_ : String
        numero_ : Integer
        superficie : Real
        nbDePieces : Integer
```

La section `columns` définit les colonnes de la relation `LesAppartements`. D'autres notations sont possibles (*documentation*).

(B) Transformation

Dans le cas où le modèle de relations est dérivé à partir d'un modèle de classe il est important de documenter le processus de transformation suivi. La section `transformation` est alors ajouté à chaque relation dérivée.

```
relation LesAppartements
    transformation
        from R_Class(Appartement)
        from R_Compo(EstDans)
        from R_OneToMany(Partage)
    columns
        nom_ : String
        numero_ : Integer
        superficie : Real
        nbDePieces : Integer
```

Dans cet exemple la transformation effectuée a été basé sur l'application de trois règles (`R_Class`, `R_Compo` et `R_OneToMany`) (*documentation*).

(C) Contraintes

Il s'agit ensuite de définir les contraintes intégrité suivantes :

- **les contraintes de domaine**. Les contraintes de domaine peuvent soit être indiquées dans le profil de la relation (par exemple `R(x:String)` ou de façon plus concise `R(x:s)`) soit être sous forme de contraintes explicites (par exemple `dom(x)=String` dans la section `constraints`) (*documentation*).
- **les contraintes de clés**. Les clés peuvent soit être définies dans le profil de la relation (par exemple `Compte(login_id)`), soit via mot clé `key` (*documentation*).
- **les contraintes d'intégrité référentielle**. Elles sont exprimées en langue naturelle ou en algèbre relationelle (*documentation*).

Se référer à la documentation de *RelationScript* pour plus d'exemples.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche bd.relations.jdd

résumé L'objectif de cette tâche est de définir des jeux de données (jdd) (positifs) pour le modèle de relations.

langage *RelationScript*

artefacts

- `bd/relations/relations.res`

(A) JDD positifs

Définir un ou plusieurs Jeux De Données (jdd) "positifs", c'est à dire respectant l'ensemble des spécifications du modèle de relations (nombre et type des colonnes, contraintes de clés, clés étrangères, contraintes provenant du domaine, etc.).

Si un ou des modèles d'objets existent, traduire en priorité ces derniers.

Les jeux de données positifs doivent être écrits en utilisant les mots clés `dataset` ou `positive dataset` du langage *RelationScript*. Ces jeux de données seront directement écrits dans le fichier `bd/relations/relations.res`.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche bd.relations.jdd.negatifs

résumé L'objectif de cette tâche est de définir des jeux de données (jdd) négatifs pour le modèle de relations.

langage *RelationScript*

artefacts

- `bd/relations/relations.res`

(A) JDD négatifs

Définir un ou plusieurs jeux de données négatifs, c'est à dire violant une ou plusieurs contraintes définies dans le modèle relationel.

Si un ou des modèles d'objets négatifs existent, traduire en priorité ces derniers.

Les jeux de données négatifs doivent être écrits en utilisant les mots clés `negative dataset` du langage *RelationScript*. Ces jeux de données seront directement écrits dans le fichier `bd/relations/relations.res`.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche bd.sql.schema

résumé L'objectif de cette tâche est d'implémenter le schéma SQL de la base de données en partant du modèle de relations.

langage SQL

artefacts

- `bd/sql/schema/schema.sql`

Introduction

Il s'agit d'implémenter en SQL le schéma de la base de données. Si un modèle de relations existe alors on cherchera a réaliser une traduction aussi fidèle et homogène que faire se peut.

Attention: Dans certains cas une base de données appelée "CyberMovies" peut être fournie à titre d'exemple. Pour vérifier si c'est le cas ouvrir ouvrir le fichier `bd/sql/schema/schema.sql` et observer son contenu. Si cet exemple n'est pas fourni, dans les tâches ci-dessous le schéma et les données devront être créés avant toute chose.

Au contraire si l'exemple CyberMovies est fourni alors les ressources associées peuvent servir à comprendre/tester la création d'une base de données, à réaliser des premières requêtes, etc. Dans ce cas il est alors fortement conseillé d'utiliser tout d'abord cette base et de lire/tester toutes les tâches `bd.sql.*` avant de commencer à écrire le nouveau schéma de données.

(A) Schéma

Implémenter le schéma relationnel en SQL revient concrètement à écrire différentes instructions `CREATE TABLE`. Ces instructions doivent être écrites dans le fichier `schema.sql`. Se référer à la documentation du SGBD utilisé pour connaître le détail de la syntaxe SQL, les types de données disponibles, la manière d'écrire les contraintes, etc.

Voici a titre d'illustration une instruction SQL de création de table.

```
CREATE TABLE Opinions(
    spectator VARCHAR(100),      -- => Spectators.name
    movie VARCHAR(100),          -- => Movies.title
    stars INTEGER,               -- BETWEEN 0 AND 5

    CONSTRAINT PK
        PRIMARY KEY (spectator, movie),
    CONSTRAINT Dom_stars
        CHECK (stars IN ('0', '1', '2', '3', '4', '5')),
    CONSTRAINT FK_spectator
        FOREIGN KEY (spectator) REFERENCES Spectators(name),
    CONSTRAINT FK_movie
        FOREIGN KEY (movie) REFERENCES Movies(title)
);
```

Note: Comme on le voit certaines normes de programmation doivent être suivies :

- tous les mots clés SQL doivent être en majuscules,
- l'indentation de 4 ou 8 espaces comme ci-dessus doit être respectée,
- les contraintes doivent être définie de manière standardisée comme ci-dessus.
  - `PK` signifie Primary Key
  - `Dom_<attributs>` pour les contraintes sur un domaine
  - `FK_<name>` pour les contraintes d'intégrité référentielle

(B) Automatisation

Un script de création `bd/sql/cree-la-bd.sh` a pour rôle d'automatiser la création de la base de données à partir du schéma. Le contenu de ce script est fourni pour le SGBD `sqlite`. Il pourra dans ce cas être utilisé tel quel. Si un autre SGBD est utilisé, ce script peut être réécrit/adapté, l'objectif étant d'avoir une seule et unique commande pour créer la base de données.

Avec sqlite entrer la commande suivante à partir du répertoire `bd/sql/`:

```
cree-la-bd.sh
```

Ce script crée une base de données vide `bd/sql/bd.sqlite3` et charge le schéma `bd/sql/schema/schema.sql`. L'exécution du script devrait ressembler à cela :

```
Nettoyage de la base de données ... fait.
Chargement du schéma ... done.
Base de données vide créée.
```

Attention: Faire attention aux éventuelles erreurs produites lors de la création. Le script ne teste pas les erreurs, elles sont simplement affichées.

Se référer éventuellement au contenu du script pour plus d'information ; pour changer par exemple la localisation de la base de données. Si un autre SGBD est utilisé le contenu de ce script devra être adapté.

(C) Vérifications

Une fois la base de données créée il est possible si on le désire d'utiliser le SGBD selectionné (ici sqlite3) pour consulter le schéma et le contenu de la base de données.

Attention: L'exemple ci-dessous suppose que l'exemple CyberMovies est disponible.

```
$ sqlite3 bd.sqlite3
SQLite version 3.22.0
Enter ".help" for usage hints.
sqlite> .tables
Cinemas     Frequents   IsOn        Movies      Opinions    Spectators
sqlite> SELECT * FROM Cinemas ;
sqlite>
```

Comme on peut le voir avec la dernière requête le contenu de la base de données est initialement vide. La *tâche bd.sql.jdd* montre comment remplir la base avec un jeux de données (jdd).

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche bd.sql.contraintes

résumé L'objectif de cette tâche est de traduire les contraintes non vérifiées nativement par le schéma SQL sous forme de triggers SQL.

langage SQL

artefacts

- `bd/sql/schema/schema.sql`

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche bd.sql.jdd

résumé L'objectif de cette tâche est de définir des jeux de données (jdd) (positifs) pour la base de données SQL.

langage SQL

artefacts

- `bd/sql/jdd/jdd*.sql`

Introduction

Des jeux de données (positifs) doivent être implémentés en SQL.

(A) Implémentation

Les jeux de données doivent être implémentés via des instructions `INSERT`. Ces instructions doivent être écrites dans les fichiers `bd/sql/jdd/jdd<N>.sql` ou `<N>` est le numéro du jeu de données.

Voici a titre d'exemple un extrait d'un jeu de données :

```
-------------------------------------------------------------------------
-- Cinemas
-------------------------------------------------------------------------

INSERT INTO Cinemas VALUES ('Hoyts CBD','Sydney');
INSERT INTO Cinemas VALUES ('Hoyts','Brisbane');
INSERT INTO Cinemas VALUES ('Event Cinema Myer','Brisbane');
INSERT INTO Cinemas VALUES ('Event Cinema','Cairns');
INSERT INTO Cinemas VALUES ('Birch Carroll and Coyles','Brisbane');
INSERT INTO Cinemas VALUES ('Event Cinema Red Center','Alice Spring');
...
```

Note: Si des modèles d'objets ou des jeux de données relationnels ont été définis auparavant ces derniers doivent être réutilisés autant que possible.

(B) Chargement

Le script de création de base de données peut être utilisé pour charger un jeu de données. Par exemple pour un jeu de données `jdd1` la création de la base de données se fait avec la commande suivante (dans l'exemple ci-dessous on suppose que le fichier `jdd/jdd1.sql` existe) :

```
cree-la-bd.sh jdd1
```

Comme on s'intéresse dans cette tâche aux jeux de données positifs, aucune erreur ne doit être détectée lors du chargement.

L'exécution du script `cree-la-bd.sh` avec le jeu de données positif `jdd1` devrait ressembler à cela :

```
Nettoyage de la base de données ... fait.
Chargement du schéma ... fait.
Chargement du jeu de données jdd1 ...fait.
Jeu de données jdd1 chargé dans la base de données.
```

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche bd.sql.jdd.negatifs

résumé L'objectif de cette tâche est de définir des jeux de données négatifs pour la base de données SQL.

langage SQL

artefacts

- `bd/sql/jdd/jddn*.sql`

(A) JDD négatifs

Cette tâche fait suite à la *tâche bd.sql.jdd*.

Contrairement aux jeux de données positifs qui ne doivent produire aucune erreur, les jeux de données négatifs doivent générer des erreurs lorsque les contraintes associées au schéma ne sont pas respectées.

Les "violations" du schéma de données doivent être documentées explicitement dans le jeu de données à l'aide d'annotations `--@ violates`. Voici par exemple un extrait du jeu de données `jddn1.sql` :

```
...
107   INSERT INTO Opinions VALUES ('Marie','The Inbetweeners 2','0');
108
109   --@ violates Opinions.PK
110   INSERT INTO Opinions VALUES ('Marie', 'The Inbetweeners 2', '3');
111
112   --@ violates Opinions.Dom_stars
113   INSERT INTO Opinions VALUES ('Marie','The Inbetweeners 2','===> VIOLATION <===
↪');
114
115   --@ violates Opinions.FK_spectator
```

(continues on next page)

(continued from previous page)

```
116   INSERT INTO Opinions VALUES ('==> VIOLATION <==','The Inbetweeners 2','0');
117
118   --@ violates Opinions.FK_movie
119   INSERT INTO Opinions VALUES ('Marie','==> VIOLATION <==','0');
120   INSERT INTO Opinions VALUES ('Adrian','The Inbetweeners 2','0');
121   INSERT INTO Opinions VALUES ('Phil','The Inbetweeners 2','2');
...
```

Chaque annotation `--@ violates` indique quelles erreurs sont censées être produites lors de l'exécution de la ligne suivante. Comme on peut le voir dans l'exemple ci-dessus le paramètre de chaque violation correspond à un nom de contrainte défini dans le schéma.

Dans l'exemple le chargement du jeu de données `jddn1.sql` produit le résultat suivant :

```
$ cree-la-bd.sh jddn1

Nettoyage de la base de données ... fait.
Chargement du schéma ... fait.
Chargement du jeu de données jddn1 ...Error: near line 23: UNIQUE constraint failed:␣
↪Movies.title
...
Error: near line 110: UNIQUE constraint failed: Opinions.spectator, Opinions.movie
Error: near line 113: CHECK constraint failed: Dom_stars
Error: near line 116: FOREIGN KEY constraint failed
Error: near line 119: FOREIGN KEY constraint failed
...
```

Le propre des jeux de données négatifs est qu'à chaque violation escomptée une erreur doit être effectivement produite. Dans l'exemple ci-dessus on retrouve les numéros de lignes où doivent se trouver les violations ainsi qu'un message d'erreur propre au SGBD.

La vérification de la correspondance entre violations escomptées et erreurs produites n'est pas automatisée. Il convient donc de vérifier "manuellement" l'alignement entre violations escomptées / erreurs produites.

La qualité d'un schéma de base de données ne tient pas uniquement en ce que ce schéma autorise mais aussi en la qualité des erreurs détectées.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche bd.sql.requetes

résumé L'objectif de cette tâche est de spécifier, d'écrire et de tester des requêtes SQL.

langage SQL

artefacts

- `bd/sql/requetes/`

Introduction

Cette tâche permet de spécifier un ensemble de requêtes à réaliser en SQL, de définir le résultat attendu et de tester que la requête fourni bien le bon résultats.

La structure de repertoire `requetes/` est le suivant :

- Les fichiers `QNNN_Identifiant.sql` contiennent les requêtes écrites en SQL.
- Le repertoire `attendu` contient pour chaque requête le résultat attendu sous forme de fichier csv.

Pour évaluer une requête `QNNN` utiliser le script suivant :

    eval.sh QNNN

Pour évaluer toutes les requêtes utiliser la commande suivante :

    eval.sh all

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

4.1.5 tâches projet.*

tâche projet.participants

résumé L'objectif de cette tâche est de définir les membres de l'équipe ainsi que leurs rôles s'ils en ont.

langage *ParticipantScript*

artefacts

- `participants/participants.pas`

Introduction

Les membres de l'équipe et leurs rôles font partie de la définition des "participants" du projet. Il s'agit de définir leurs caractéristiques principales dans le fichier `participants/participants.pas`. Se reporter à la documentation de *ParticipantScript* pour plus d'information.

(A) Rôles

Note: Dans certains projets aucun rôle n'est prédéfini et ceux-ci, s'ils émergent au cours du projet sont définis au fil de l'eau. Cette tâche doit dans ce cas être executée "à la demande" plutôt qu'en début de projet.

Définir, s'ils sont connus, les roles que peuvent jouer les membres de l'équipe en utilisant les mots clés `team role`. Un projet peut par exemple définir un rôle `IntegrationManager`. Indiquer brièvement quelles sont les fonctions associées à chaque rôle. Des rôles "externes" à l'équipe peuvent être définis, par exemple `ConsultantBD` pour un expert en base de données donnant des conseils sur les base de données et/ou réalisant des audits sur ce thème.

Il est à noter qu'un rôle peut être joué par plusieurs membres et inversemment qu'un membre peut jouer plusieurs rôles. De même une personne peut jouer un rôle à un moment donné puis changer de rôle par la suite. Tous ces rôles seront associés à la personne en mentionnant si nécessaire ces changements la description de la personne. Par exemple `| Integration Manager du 12/03/2020 au 17/03/2020.`

Note: Dans le cadre d'un projet "exploratoire", du point de vue de la méthodologie, les rôles peuvent être définis au fur et à mesure du projet,

---

(B) Personnes

Les membres de l'équipe doivent être brièvement décrits : nom et trigramme. Utiliser le mot clé `person`. Indiquer le ou les rôles que chaque personne a ou va jouer. Définir également les éventuels consultants extérieurs ou toute autre personne impliquée dans le projet, dans son évaluation, etc.

(Z) Suivi et status

Suivi: Des questions ou des hypothèses ? Voir la *tâche projet.suivis*.

Status: Avant de terminer cette tâche écrire le status. Voir la *tâche projet.status*.

tâche projet.planning.gantt

résumé L'objectif de cette tâche est de planifier la suite du projet à l'aide de diagrammes de gantt.

langage Gantt

résultats

- `projet/sprint<N>/provisionnel/*`

Introduction

Dans le cas de cycles de vies séquentiels (cascade ou V) ou incrémentaux l'utilisation de diagrammes de gantt est classique. Il s'agit de définir *en avance*, autant que faire se peut, les tâches à réaliser. Planifier consiste à affecter à ces tâches des ressources, dont des ressources humaines, ainsi qu'à répartir les tâches dans le temps, en estimant, entre autre, la durée des tâches.

Ici, l'outil open source gantt project sera utilisé.

La planification du projet, se fera en début de chaque sprint ou incrément. Contrairement à ce qui se fait dans le cadre de méthodes agiles la planification devra couvrir la période du début du projet *jusqu'à la fin du projet*, donc au delà de la fin du sprint/incrément à venir. Bien évidemment ce dernier sera plus détaillé.

Un "planning prévisionnel" est défini au début de chaque sprint/incrément. Un "planning effectif" est établi à la fin de chaque sprint/incrément. Voir la *tâche projet.planning.effectif*. **Dans cette tâche on s'interesse au planning prévisionnel.**

Dans le cadre d'une gestion de projet traditionnelle, c'est le chef de projet qui assure la planification du projet ainsi que son suivi.

Note: Le resultat de cette tâche sera déposé dans le répertoire `projet/sprint<N>/plannings/previsionnel` où `<N>` est le numéro du sprint.

Lancer gantt project (il doit au préalable avoir été installé) :

```
ganttproject
```

Ouvrir le fichier correspondant à l'incrément concerné. Par exemple pour le sprint 1, ouvrir le fichier suivant :

```
projet/sprint1/plannings/previsionnel/planning-previsionnel.gan
```

Le reste de cette tâche va consister à définir le planning prévisionnel, et donc à compléter ce fichier.

Note: Si un projet initial est fourni il s'agit de le compléter ou de l'adapter selon les cas. Certaines tâches présentées ci-dessous peuvent ne pas être nécessaires.

Note: Ci-dessous on suppose que l'interface graphique de gantt project est en Français. Pour choisir la langue utiliser le menu `Edit > Settings > Application UI > Language`.

(A) Calendrier

La première tâche consiste à définir le calendrier du projet, c'est à dire le début, la fin du projet, les jours travaillés, jours fériés, vacances, examens, etc.

Utiliser pour cela le menu `Projet > Paramètres du projet > Calendrier`.

(B) Jalons

Les dates des différents jalons (milestones) du projet doivent être définies. Ces jalons correspondent par exemple aux livraisons, audits et à la soutenance. Ajouter tous les événements connus et dont la date est fixe et défnie.

Dans gantt project un jalon est un cas particulier de tâche. La création d'un jalon se fait en deux temps :

- créer une tâche avec `Ctrl T`
- transformer cette tâche en jalon avec `Alt Enter > Général > Point bilan`.

(C) Ressources

Dans le cadre de la gestion de projet traditionnelle, les membres de l'équipe de dévelopement sont considérées comme des "ressources" ; plus particulièrement des "ressources humaines". Dans le cadre de gantt project ces ressources doivent être déclarées car elles vont être affectées aux tâches. Utiliser l'onglet `Resource Chart` pour voir l'ensemble des ressources.

Utiliser le menu `Ressource > Nouvelle ressource (Ctrl H)`. Utiliser le trigramme de chaque membre en lieu et place du nom.

(D) Tâches

Dans un premier temps lister les tâches à réaliser *sans chercher à les ordonnancer*. Ceci se fera dans une étape utltérieure. Utiliser dans un premier temps `Ctrl T` pour introduire rapidement les différentes tâches. Ne pas chercher à déterminer la durée de chaque tâche. Lister simplement les tâches.

Lorsque les tâches font références à des tâches ModelScript utiliser leur idendificateur (par exemple `projet.planning.gantt`). Ajouter, lorsque nécessaire, un prefixe (par exemple `sprint2.projet.planning.gantt`).

(E) Décomposition

La granularité des tâches à prendre en compte dépend du projet. Définir des tâches trop fines risquent d'être trop lourd. Cela rend la gestion de projet inefficace dans la mesure où trop de tâches doivent être planifiées. L'unité de gantt project (ainsi que d'autres logiciels similaires) est le jour. Une tâche d'une durée inférieure à 1 jour devra *peut être* être regroupée avec d'autres tâches (voir ci-dessous). Dans tous les cas de figures, planifier un projet à la journée près est déjà une complexe.

Les tâches peuvent être emboitées, par exemple pour décomposer une tâche abstaite en tâches concrètes. Utiliser `Alt flêche->` pour imbriquer une tâche dans une autre.

Le nom des tâches utilisées peut *éventuellement* servir pour regrouper certaines tâches (par exemple `bd` pourrait regrouper les tâches `bd.sql.jdd` et `bd.sql.schema`). Cette solution n'est cependant pas toujours la meilleure. Il peut être préférable de grouper des tâches par incréments ou autre.

(F) Affectation

Un ou plusieurs membres de l'équipe de développement peuvent être affecté à une tâche, et avec une quotité éventuellement inférieure à 100%. Par exemple `NZN` peut être affecté à la tâche `bd.sql.schema` à 50%.

Pour réaliser cette affectaton avec gantt project utiliser `Alt Enter > Ressources > Ajouter`. Il peut être utile de définir un référent ou responsable pour la tâche. Utiliser dans ce cas la case à cocher `Responsable`.

L'affectation des ressources doit être faire conjointement avec la planification. Voir ci-dessous.

(G) planification

Une fois les tâches et les ressources définies il s'agit de réaliser la planification, c'est à dire :

- affecter des ressources aux tâches (voir ci-dessus).
- établir la durée prévue pour chaque tâche,
- définir les éventuelles dépendances entre tâches,
- définir la date de départ de chaque tâche.

Le résultat de ces différentes opérations permet de définir un planning prévisionnel et de "caler" chaque tâche dans le temps.

Dans gantt project les propriétés d'une tâche peuvent facilement être modifiées en tapant `Alt Enter`. Il est ensuite possible de définir le nombre de jour estimé ainsi que les ressources associées.

La durée des tâches dépend évidemment des ressources associées. Les dates de début dépendent des dépendances entre les tâches et de la durée des tâches. La planification est donc un exercice difficile car différentes variables doivent être prises en compte simultanément.

Dans le cadre d'une gestion de projet traditionnelle c'est le chef de projet qui gére le planning du projet.

(H) Diagramme de gantt

Après avoir réalisé la planification faire une copie d'écran du diagramme de gantt. Modifier au préalable les paramètres d'affichage. Utiliser pour cela le menu `Edition > Préférence` puis l'onglet `Propriétés du diagramme de Gantt`, en bas d'écran la section "Détails". Faire afficher les noms des ressources ainsi que le nom des tâches plutôt que leur id. Créer une vue globale du diagramme (fichier `diagrammes/plan.gan.png`) et éventuellement une ou plusieurs autres vues plus détaillées (fichier `diagrammes/<NOM>.gan.png` ou `<NOM>` est le nom de la vue).

(I) Diagramme des ressources

Créer un diagramme des ressources. Utiliser pour cela l'onglet `Diagramme des Ressources` sur l'écran principal et immédiatement au dessus de la liste des tâches. Faire une copie d'écran correspondant à la vision globale (fichier `diagrammes/plan.res.png`) accompagnée éventuellement d'une ou plusieurs vues d'intérêt `diagrammes/<NOM>.res.png` ou `<NOM>` est le nom de la vue)

tâche projet.planning.agile

résumé L'objectif de cette tâche est de planifier les tâches à effectuer dans le cadre d'un sprint.

langage GitHub

résultats

- `github:<project>`
- `github:<issues>`
- `github:<milestones>`
- `projet/sprint<N>/provisional-plan/*`

Introduction

L'activité de planification est beaucoup plus légère dans le cadre de projets en mode agile que dans les projets séquentiels (voir la *tâche projet.planning.gantt*). Il s'agit ici de définir d'incrément en incrément les tâches à réaliser.

Dans cette tâche GitHub sera utilisé, d'une part pour représenter les tâches via des "issues", d'autre part en utilisant la notion de "projet" pour suivre l'état d'avancement du projet.

(A) Jalons

Définir les différents jalons (milestone) du projet (livraisons, audits, soutenance, etc.). Utiliser pour cela les "Milestones" GitHub. Aller sur le dépot de groupe, puis `Issues > Milestone > New milestone`.

(B) Projet

Créer un tableau de bord (appelé "project" dans GitHub) associé au dépot de groupe. Utiliser l'onglet `project` du dépot puis `Create a project`. Définir les colonnes en fonction des besoins.

Note: Les colonnes pourront être ajoutées par la suite au fur et à mesure des besoins.

Ce tableau de bord pourra contenir des tâches sous forme d'issues, mais également de "simples" notes. Il est possible de définir plusieurs tableaux, par exemple pour plusieurs sprint ou pour des "projets" s'exécutant en parallèle. Définir la structure la plus simple possible mais adaptée au besoin de l'équipe. Cette structure pourra être adaptée au fil de l'eau.

(C) Tâches

Il s'agit ici de définir les tâches du projet sous forme d'issues GitHub. On cherche plus particulièrement a **établir la traçabilité** entre :

- (1) le processus ModelScript
- (2) le processus suivi effectivement.

Lorsqu'une tâche est dérivée directemment d'une tâche de référence ModelScript le nom de l'issue y fera référence directement. Par exemple la tâche `sprint2.projet.planning.gantt` indiquera qu'il s'agit d'effectuer la tâche `projet.planning.gantt` pour le `sprint2`. De même on pourra utiliser un nom comme `s1,d1,d2.concepts.classes` pour la tâche consistant à compléter le modèle de classes avec le scénario `s1` et les incréments `d1` et `d2`.

Note: GitHub permet de créer des références entre issues, sans pour autant donner de recommendations sur la manière d'utiliser cette fonctionnalité. Ci-dessous cette l'utilisation de cette fonctionnalité est formalisée afin d'assurer la traçabilité entre issues.

(D) Références

Utiliser le nom de la tâche pour référencer la tâche "mère" n'est pas toujours facile ni souhaitable. Dans le cas de relation "mère - fille" on utilisera par contre de manière systèmatique le mot clé `org` suivi du numéro d'issue de la tâche mère. Par exemple si une tâche `#68` dérive de la tâche `#34` alors le corps de la tâche `#68` débutera par :

```
org #34
```

Les tâches peuvent ainsi être imbriquées. Si une tâche ne correspond à aucune tâche ModelScript alors ajouter le label `Extra` à cette tâche pour mettre en l'avant qu'elle sort du processus. Voir la section suivante pour les labels.

L'exemple ci-dessous montre un arbre des tâches avec la tâche principale `concepts.classes` étiquettée `ModelScript` car il s'agit d'une tâche de référence du processus. Les tâches `#34`, `#68` et `#122` sont elles des sous-tâches. La tâche `#79` n'a pas de tâche parente. Elle a donc le label `Extra` pour indiquer que cette tâche échappe au processus ModelScript.

```
#17 concepts.classes [ModelScript]
    #34 s1,d1-3.concepts.classes     org #17
        #68 Ajouter la notion d'employés et de bureaux    org #34
        #122 Ajouter la notion d'envois    org #34
#79 Faire de l'espace sur le disque    [Extra]
```

(E) Labels

Associer à chaque tâche les "labels" correspondant. Par exemple "bd". Pour cela utiliser `Labels` dans le panneau de droite de l'issue concernée.

(F) Planification

Pour estimer la complexité des tâches à réaliser l'une des techniques est de jouer au "poker planning". Réaliser une telle session avec l'ensemble des membres de l'équipe. Définir ensuite à quel jalon telle ou telle tâche doit être associée.

Une fois la liste des tâches établie définir à quel jalon l'issue doit être associée. Cela permet de définir dans quel incrément la tâche doit être réalisée. Pour définir le jalon associé à une tâche utiliser la section `Milestone` à droite d'issue à assigner.

(G) Affectations

Les tâches peuvent être affectées à une ou plusieurs personnes. Utiliser pour cela la section `Assignee`. Contrairement aux projet en mode séquentiel, dans un projet en mode agile il n'est pas nécessaire de réaliser cette affectation a priori et au début du projet. Il est classique d'avoir un lot de tâches non assignées et qu'un membre de l'équipe se saisisse à un moment donné d'une tâche.

(H) Tableau

Faire une copie d'écran du tableau ("project" github) en début de projet et la ranger dans `provisional-plan/diagrammes/plan.github.png`.

tâche projet.standup

résumé Cette tâche consiste à réaliser chaque jour un "standup meeting" ou "daily scrum meeting".

Introduction

Le "daily scrum meeting" (ou "standup meeting") est l'un des rituels les plus populaires associé à la méthode scrum.

Note: L'issue `projet.standup` dans le dépot de groupe sera utilisée pour le suivi des standup meetings. Différents commentaires seront ajoutés dans cette issues au fur et à mesure de l'avancement du projet. Voir ci-dessous pour plus de détails.

(A) Patrons

Contrairement à ce que l'on peut imaginer les standup meetings sont bien plus que des meetings debout. Ces réunions peuvent et doivent être organisés autour d'un certain nombre de patrons. Si un scrum master a été nommé, l'article "It's Not Just Standing Up: Patterns for Daily Standup Meetings" (article) est une lecture de choix. L'article peut être survolé. La section "Patterns of daily stand-up meetings" est la plus intéressante. Cette section donne des indications sur la manière de gérer les standup meetings.

(B) Horaire

Il est important de choisir un horaire fixe pour les standup meeting, en début de journée. Indiquer cet horaire sous forme de commentaire dans l'issue `projet.standup` du groupe. Si a un moment donné l'horaire est changé il est impératif d'indiquer ce changement. Donner les raisons de ce changement ainsi que le nouvel horaire.

Attention: Le standup meeting doit démarrer toujours à l'heure prévue, même si un membre du groupe est en retard. Les retards doivent être exceptionnels.

(C) Présence

Il est absolument indispensable que tous les membres du groupe assistent systématiquement et sans exception à tous les standup meetings.

(D) Standup meetings

Les standup meetings doivent avoir lieu de manière *systématique* chaque jour, sauf les jours où d'autres événements ont lieu (audits par exemple). Chaque jour, le commentaire *"fait, durée MM"* sera ajouté à la liste des commentaires, indiquant simplement que la réunion a été faite et à duré environ MM minutes.

Attention: RAPPEL: les standup meetings ne doivent pas excéder 15 minutes, mais en revanche ils doivent être fait de manière systèmatique.

(E) Empêchements

Si un empêchement ("impediment" dans la terminologe scrum) survient et qu'il ne peut pas être résolu immédiatement, le noter dans le modèle de suivi (`projet/suivis/suivis.trs`). Utiliser le mot clé `impediment` en *TrackScript*.

(F) Discussions

Les discussions entre deux ou plusieurs membres du groupe sont à proscrire. L'idée du standup meeting n'est pas de résoudre des problèmes mais plutôt d'informer chaque membre du groupe de l'état d'avancement des différentes tâches ainsi que de donner une vision sur ce qui va être accompli dans la journée. Si des dicussions sont nécessaires celles-ci peuvent être tenues immédiatement après le standup meeting ou à tout autre moment.

Utiliser le modèle de suivis (`projet/suivis/suivis.trs`) pour consigner en *TrackScript* les décisions prises et pour modéliser un éventuel plan d'action.

tâche projet.suivis

résumé Cette tâche a pour objectif de prendre connaissance du modèle modèle de suivis et de son utilité.

langage *TrackScript*

artefacts

- `suivis/suivis.trs`

Introduction

Tout au long du cycle de vie d'un projet il est souhaitable de consigner par écrit différents points de suivis ;

- questions,
- hypothèses,
- empêchements,
- etc.

L'objectif de cette tâche est de s'assurer que le modèle de suivis est compris.

(A) TrackScript

Lire la page décrivant le langage de *TrackScript*.

Tout au long du projet consigner dans le modèle de suivis toutes les les points de suivis apparaissant au fil de l'eau du projet.

tâche projet.status

A status report have to be written down for each model or WorkDefinition. In order to do so a file `status.md` is associated to each model, or to a group of models. There is as well a global `status.md` file at the top level directory.

Rules

The following information should be present in the status file:

- what have been done,
- what is partially working and have to be continued / improved,
- what remains to be done.,
- a short synthesis.

Add other information when necessary (for instance bug found or issues encountered during the realisation of the task, a reference to an important hypothesis).

Be concise, yet as structured as possible.

If everything is ok, then a simple message could be enough. If some parts remain to be done an indicator such as "(about) 25% done" can be helpful. If the task has subtasks this could be "T3: Shape, Rectangle, Circle to be implemented".

Be concise, yet as structured as possible.

tâche projet.planning.effectif

résumé L'objectif de cette tâche est d'établir d'une part un planning intermédiaire d'autre un planning effectif en fin d'incrément.

langages Gantt, GitHub

résultats

- `projet/sprint<N>/effective-plan/*`

Introduction

Les plannings prévisionnels effectués en début de chaque d'incrément/sprint sont basés sur les prévisions du déroulement des tâches à venir. Ici il s'agit ici au contraire d'effectuer :

- un bilan intermédiaire, par exemple en milieu du déroulement d'un incrément,
- un bilan en fin d'incrément/sprint.

Ces bilans rendent compte du déroulement effectif des tâches déjà réalisées et montrent de manière prévisionnelle le planning ajuster pour la suite du projet. Le bilan en fin d'incrément sera utilisé pour établir le bilan prévisionnel au début de l'incrément suivant.

(A) Planning intermediaire - GitHub

Si le mode agile a été selectionné, les tâches doivent avoir été déplacée tout au long de l'incrément d'une colonne à l'autre du tableau, pour refléter l'état d'avancement du projet. Réaliser simplement une copie d'écran de ce tableau en milieu de sprint par exemple. Sauvegarder cette copie dans le fichier `projet/sprint<N>/effective-plan/diagrammes/intermediate-plan.github.png`.

Note: Il s'agit ici simplement de rendre compte du déroulement de l'incrément en cours d'incrément. L'image pourra être utilisée dans une présentation pour montrer l'incrément "en cours de vie" avec des issues dans des colonnes "à faire" et "fait", mais aussi dans les colonnes intermédiaires (ce qui n'est pas le cas des autres plannings dans lesquels les colonnes de début ou de fins sont parfois les seules remplies).

(B) Planning intermediaire - Gantt

Si le modèle de gantt est utilisé, faire le bilan sur les tâches effectuées. Utiliser la possibilité de définir l'état d'avancement d'une tâche. Soit en utilisant le champ `Avancement` dans le panneau de propriété de tâche (`Alt Enter`), soit en déplacant le curseur de gauche à droite directement sur le diagramme de gantt. L'avancement de la tâche est représentée par un trait noir à l'interieur de la tâche. Sauvegarder le modèle intermédiaire dans `projet/sprint<N>/effective-plan/intermediate-plan.gan`. Réaliser une copie d'écran dans `projet/sprint<N>/effective-plan/diagrammes/intermediate-plan.gan.png`.

(C) Planning effectif - GitHub

Si GitHub est utilisé réaliser en fin de sprint la même opération que pour le plan intermédiaire. Sauvegarder l'image du tableau dans le fichier `projet/sprint<N>/effective-plan/diagrammes/plan.github.png`

(D) Planning effectif - Gantt

Si le modèle de gantt est utilisé réaliser en fin d'incrément/de projet la même opération que pour le plan intermédiaire. Sauvegarder le version finale du plan dans `projet/sprint<N>/effective-plan/plan.gan` et une copie d'écran dans `projet/sprint<N>/effective-plan/diagrammes/plan.gan.png`.

tâche projet.retrospective

résumé Cette tâche vise a réaliser une rétrospective portant sur le dernier sprint/incrément écoulé.

artefacts

- `projet/sprint<N>/retrospective/*`

Introduction

Réaliser une rétrospective en fin d'un sprint, juste avant l'audit. Il s'agit d'examiner de manière rétrospective et introspective le déroulement du dernier sprint. L'objectif est de se concentrer sur la méthodologie et les pratiques utilisées et non pas sur le produit réalisé. Autrement dit les retrospectives portent sur le processus, pas sur le produit. Dans une rétrospective il s'agit de discuter de la "manière de faire", des bonnes ou mauvaises pratiques, de l'utilisation ou de la mise en place d'outils, etc.

(A) Retrospective

Réaliser une séance de rétrospective avec l'ensemble des membres de l'équipe. L'une des manière de mener une rétrospective consiste à répondre à trois questions :

- **ce qui n'a pas marché ?** (en terme de processus). Le scrum master doit faire attention à ce que les aspects négatifs associés à cette question ne prennent pas trop d'importance dans la réunion.
- **ce qui à bien marché ?** (en terme de processus). Mettre l'accent sur ce point.
- **ce qui pourrait être essayé pour améliorer ?** (en terme de processus). Ce dernier point est primordial car il dégage des axes d'amélioration et se tourne vers le futur. C'est l'un des objectifs principal de mener une rétrospective.

(B) Compte rendu

Les conclusions de la rétrospective doivent être consignées dans le fichier `projet/sprint<N>/retrospective/retrospective.md` sous forme d'une simple liste de points. Aucune rédaction n'est nécessaire mais le contenu doit être présent.

Ce document servira de base pour les audits.

Organiser ce document en trois sections selon les trois questions mentionnées ci-dessus.

tâche projet.livraison

résumé L'objectif de cette tâche est de livrer le logiciel.

artefacts

- `github:release`
- `CHANGES.txt`

Introduction

La livraison est une opération formelle (qui peut être contractuelle) dans laquelle l'équipe de développement délivre l'état le plus avancé du logiciel au client. Le logiciel passe typiquement d'un environement de développemment à un environnement de pre-production ou de production.

(A) Tests

Avant de livrer un logiciel il est bien sûr nécessaire de s'assurer que la version la plus stable et la plus à jour est mise à disposition du client. Toute livraison doit être précédée par des tests poussés.

(B) Notes de livraison

Les "notes de livraison" (releases notes) sont d'une importance capitale dans le processus de livraison. Les notes de livraison constituent la première source d'information pour un client pour déterminer si les modifications/évolutions demandées ont été implémentées, si les bugs ont été corrigés, etc. La satisfaction du client dépend du contenu de ses notes. Il va sans dire que les notes de livraisons doivent être fidèles au contenu de la livraison.

Les notes de livraisons doivent donc être rédigées avec le plus grand soin. Il peut être utile de consulter les messages de "log" de git pour être sûr de ce qui a été modifié. Utiliser la commande `git log`.

Concrètement les notes de livraison seront associées à la livraison (release GitHub). Voir ci-dessous. Le projet bootstrap fourni des exemples de notes de livraison.

(C) Livraison

Dans GitHub la livraison du logiciel se fait via le bouton `Releases` (suivi du nombre de releases) sur la page principale du dépot du groupe. Utiliser ensuite `Create a new release`. A la fin du sprint 2, utiliser "v2.0" comme "tag". Attacher les notes élaborées dans la section ci-dessus. Dans un premier temps indiquer qu'il s'agit d'une pré-release.

Si pour une raison ou pour une autre une livraison mineure devait être produite avant la date de livraison, utiliser un numéro de version tel que "v2.1".

Immédiatement avant la livraison décocher la case `this is a pré release` de la derniere livraison.

tâche projet.audit

résumé L'objectif de cette tâche est (1) de préparer l'audit, (2) de réaliser cet audit, puis (3) d'en faire la synthèse.

artefacts

- `projet/sprint<N>/audit/*`

Introduction

L'objectif d'un audit est de faire le bilan, le plus objectif possible, des résultats obtenus pendant un incrément ainsi que du processus méthodologique menant à ces résultats.

Il s'agit pour l'équipe de développement d'indiquer :

- ce qui a été fait, doit être amélioré, reste à faire, (se baser sur les fichers `status.md`),
- quels résultats ont été produits,
- quelles tâches ont été réalisées,
- quelles difficultés ont été rencontrées,
- quels empêchements bloquent ou freinent l'avancée du projet.

Il ne s'agit pas de "vendre" ce qui a été fait en en exagérant les mérites, mais plutôt de convaincre que ce qui a été fait est solide et que l'équipe est suffisemment fiable pour mériter l'octroi des ressources nécessaires à un nouvel incrément.

L'objectif de l'audit lui-même est d'intéragir avec le comité d'audit, de l'informer, mais aussi de recueillir les recommandations émises afin d'établir un rapport d'audit suivi d'actions précises.

Note: En Scrum le "sprint review" et la cérémonie correspondant le plus aux audits.

(A) Transparents

Chaque audit est basé sur une présentation effectuée à base de transparents. La dernière version doit être convertie en fichier .pdf dans `projet/sprint<N>/audit/audit.pdf`

(B) Contenu

La présentation doit être basée sur :

- les différentes captures d'écran réalisées au cours du sprint (plannings, diagrammes de classes, tableau GitHub etc.),
- les différents fichiers produits pendant le sprint,
- les différentes tâches ModelScript réalisées.

Il doit être possible, pour chaque transparent, de savoir à quel artefact et/ou quelle tâche, le transparent fait référence. Voir la section "Traçabilité" ci-dessous.

Si une rétrospective récente à eu lieu faire part des résultats de cette rétrospective dans la présentation.

(C) Suivis

Les éléments du modèle de suivis doivent être utilisés dans la présentation pour montrer quelles décisions ou hypothèses ont été faites, quelles questions sont à l'ordre du jour et quels empêchements ont freiné le projet. Faire référence à chaque suivi par son identifiant (par exemple `Q3`) et par son titre.

(D) Glossaire

Les transparents doivent impérativement faire référence aux termes du glossaire. Lorsque possible utiliser les backquotes "`" pour faire référence à ces termes.

(E) Traçabilité

Chaque transparent, chaque élément de présentation doit faire référence, autant que faire se peut, aux entités définies dans les modèles ou plus généralement dans le projet. Faire référence aux scénarios (p.e. `S1`), aux incréments (p.e. `I3`), aux questions (p.e. `Q2`), aux tâches (p.e. `concepts.glossaires`), aux issues GithHub (p.e. `#12`), etc. Faire référence aux artefacts par leur nom court (p.e. `classes.cl1`).

Attention: La possibilité d'identifier de manière précise "de quoi parle" chaque transparent, chaque phrase, chaque image est un critère important d'évaluation. Chaque transparent devrait donc comporter plusieurs voire de nombreuses références.

Il peut être utile, mais pas indispensable ni facilement faisable, de donner un document à l'auditoire indiquant à quoi correspondent les références principales. Dans tout les cas, quelqu'un lisant les transparents avant ou après la soutenance devrait pouvoir retrouver tous les éléments cités.

(F) Présentation

Lors de la présentation effective, c'est la dernière version des transparents sur GitHub qui doit être présenté.

Pendant la présentation chaque membre du groupe doit parler et répondre aux questions qui lui sont posées (les questions peuvent être "nominatives")

Un ou deux "secrétaires" doivent être nommés afin de prendre des notes tout au long de l'audit. Il peut être intéresant d'avoir deux secrétaires pour avoir "plus de notes". Ce peut être utile lors d'échanges rapides avec l'auditoire. Le deuxième secrétaire est là aussi pour se substituer au premier lorsque celui-ci intervient.

Les notes prises pendant l'audit serviront de résumé d'audit.

Attention: Perdre des informations ou remarques faites pendant l'audit est une faute grave. Aucun client n'apprécie d'avoir à redire une fois de plus ce qui a été déjà dit lors d'une précédente réunion. Cela démontre un manque de professionalisme.

(G) Démonstration

Une démonstration du produit pendant l'audit est la bienvuenue. Cependant une telle démonstration n'a pas de caractère obligatoire.

Se reporter à la section *Démonstration* de la tâche `projet.soutenance` pour plus de détails sur la manière d'organiser une présentation.

Note: Les démonstrations d'audits n'ont pas le caractère formel que l'on trouve dans la démonstration de soutenance. Certains élements mentionnés pourront donc être simplifiés.

Si une démonstration est faite pendant une audit et qu'une autre démonstration a été faite précédemment, il est judicieux de montrer de manière explicite les différences entre les fonctionnalités successives. Ceci peut se faire sous la forme de phrases comme "Avant ici il y avait . . . ".

(H) Documents

Il peut être utile (mais en général pas nécessaire) de distribuer aux membres du comité d'audit des documents. C'est le cas notamment si certains transparents sont difficilement lisibles (p.e. les diagrammes de classes ou modèles de tâches).

(I) Compte rendu

Après l'audit faire tout d'abord un débriefing entre les membres de l'équipe.

Etablir ensuite un compte rendu faisant état des principales remarques faites lors de l'audit, suivies des actions à entreprendre. Le compte rendu d'audit doit se faire immédiatement après l'audit, au moins pour la partie "remarques effectuées".

Le compte rendu doit être réalisé sous forme de texte dans le fichier `projet/sprint<N>/audit/resume.md`. Il peut s'agir simplement de quelques lignes. Utiliser un style télégraphique, une liste de points. Il ne s'agit pas d'un document formel mais simplement d'un mémo principalement à destination de l'équipe. La forme n'est pas primordiale mais le contenu est par contre particulièrement important car c'est lui qui défini l'orientation du prochain sprint.

Attention: Si des décisions importantes ont été prises, les consigner dans le fichier `suivis/suivis.trs`.

tâche projet.soutenance

résumé Cette tâche vise a définir les tâches liées à la préparation de la soutenance ainsi qu'à la soutenance elle même.

Introduction

Contrairement aux audits qui peuvent être considérés comme des réunions de travail avec le comité d'audit, la soutenance vise à présenter à l'auditoire (qui peut être plus large) les mérites du travail réalisé, mérites en termes de :

- développement,
- conception,
- méthodologie.

Il ne s'agit pas de montrer que l'équipe de développement est arrivée à réaliser une "démo" qui semble marcher, mais plutôt que le produit est de qualité, qu'un processus clair et solide a été suivi et que le logiciel a été conçu dans les règles de l'art.

Il s'agit de donner de la confiance ; de donner à l'auditoire confiance dans ce qui a été réalisé, avec l'eventuelle envie de continuer de travailler ensemble sur des extensions du projet, voire sur de nouveaux projets.

Note: La cérémonie Scrum qui s'approche le plus de la soutenance et le "Sprint Review".

(A) Fichiers

La dernière étape du dernier sprint correspondant à une soutenance plutôt qu'à un audit. Les noms de fichiers doivent éventuellement être ajustés comme ci-dessous.

Le répertoire à utiliser est le répertoire `projet/sprint<N>/soutenance` où `<N>` est le numéro du dernier sprint. Si ce répertoire n'existe pas renommer le répertoire `audit` en `soutenance`. Faire de même pour les fichiers se trouvant dans le répertoire.

Le fichier `resume.md` peut être éliminé car la soutenance ne donne pas lieu à compte rendu.

(B) Transparents

La version finale des transparents, en pdf, doit se trouver dans `projet/sprint<N>/soutenance/soutenance.pdf`. La version présentée lors de la soutenance doit impérativement correspondre à la version présente sur GitHub.

(C) Contenu

Comme pour les audits, les transparents de la soutenance doivent faire explicitement référence aux différents artefacts créés (plannings, diagrammes de classes, modèles de tâches, etc.). Les résultats obtenus doivent être clairement mis en avant et il est indispensable de faire référence explicitement aux scénarios, aux incréments, etc.

(D) Traçabilité

La traçabilité dans la conception et le développement est un des critères important de l'évaluation. Voir la section *Traçabilité* de la *tâche projet.audit* pour plus de détails.

(E) Démonstration

Une démonstration du produit devra être faite. Cette démonstration doit **impérativement être préparée, pas à pas**. La démonstration peut être intégrée dans la présentation où faite dans une partie spécifique.

Il peut être utile de fournir à l'auditoire un "script" de la démonstration permettant de montrer ce qui est démontré, étape par étape, en termes d'objectifs, de résultats attendus, d'interactions, etc.

Le contenu de la démonstration doit également être enoncé à l'oral avant le déroulement des actions.

Le contenu global de la démonstration devra être établi plusieurs jours avant la soutenance, de manière à définir le script, les données utilisées dans la démonstration, les personnes impliquées, etc.

La démonstration peut éventuellement être faite en deux sessions :

- une première session, avec un scénario figé et bien délimité correspondant à un déroulement normal d'un utilisateur. Cette session montrera un scénario "métier" sans trop rentrer dans les détails techniques. Elle doit raconter une histoire bien identifiée.
- une deuxième session plus "ouverte", plus "technique", plus "exploratoire", c'est à dire en suivant des chemins qui ne seraient normallement pas suivis par un utilisateur standard. Ce peut être par exemple pour montrer des scénarios d'erreurs ou des détails techniques jugés important.

Répeter impérativement la démonstration, en faisant attention notamment à ne pas aller trop vite, en s'assurant que le niveau de la voix du présentateur soit audible, etc. A tout moment, l'auditoire doit être en mesure de comprendre "ce qui se passe et pourquoi cela se passe ainsi". Lors d'une répétition il peut être utile qu'une personne extérieure ou qu'un autre membre du groupe assiste à la démonstration et fasse part de ses commentaires.

Les jeux de données utilisés dans la démonstration devront correspondre aux jeux de données élaborés en début de cycle de vie. Ces jeux de données peuvent par exemple provenir des modèles d'objets (`concepts/objets/`) ou modèle de scénarios (`cu/scenarios/`).

Dans un cycle de vie en V, l'exécution de scénarios prédéterminés et définis en début de projet peut constituer un "test de recette", c'est à dire un test qui détermine l'acceptation ou non de la livraison du projet. Montrer en quoi le ou les scénarios exécutés divergent ou non des scénarios prévus initialement.

Pendant la démonstration, attention à utiliser au maximum les termes métiers définis dans le glossaire. Le narrateur peut jouer le rôle d'un utilisateur en disant par exemple *"Je suis Paul, un bibliothécaire et je veux ... Maintenant j'ai besoin de ... et je fais ... "*, etc.

Si certains détails ne peuvent pas être montrés pendant la démo n'hésitez pas à dire *"pour ceux intéressés par ce point là, nous pourrons y revenir pendant la scéance de questions"*. Cela permet entre autre de diriger les questions vers des éléments qui seront déjà préparés.

Faire quelques copies d'écrans et les intégrer en fin de présentation pour palier d'éventuelles difficultés à dérouler la demonstration.

(F) Documents

Comme pour les audits il peut être utile de distribuer des documents aux membres du jury. C'est le cas notamment d'informations relatives aux scénario(s) suivi(s) dans la démonstration. Tous les documents permettant de mieux suivre la démonstration ou la soutenance seront les bienvenus.

(G) Soutenance

Déterminer avant la soutenance sur quel machine la présentation et la démonstration vont être faites. Vérifier **avant** la soutenance que les problèmes de connections sont résolus. Prévoir éventuellement une machine de repli.

Tous les membre du groupe doivent parler.

Chaque membre du groupe doit parler et répondre aux questions qui le concerne.

4.1.6 tâches dev.*

tâche dev.general

résumé Cette tâche regroupe l'ensemble des activités de développement non couvertes par une autre tâche dev.*.

artefacts

- `dev`

Introduction

Cette tâche générique sert de containeur pour toutes les tâches de développement n'ayant pas été répertoriées par ailleurs.

CHAPTER 5

Tools

5.1 Tools

In the current version, tooling is minimal. Consistency checking relies on the `use` tool. The `modelc` compiler is not available yet. Syntax highlighting for gedit is under construction. Check next versions for improved tool support...

5.1.1 modelc

The modelc compiler is not available yet. Check further versions for availability.

5.1.2 USE

The current version of ModelScript depends on the USE tool from the university of bremen. USE allows to check consistency between object and class models, as well as drawing UML diagrams.

Checking classes consistency:

```
use -c concepts/classes/classes.cl1
```

Checking classes/objects consistency:

```
use -qv concepts/classes/classes.cl1 concepts/objets/o1/o1.ob1
```

Creating a class diagram using the graphical interface:

```
use concepts/classes/classes.cl1 concepts/objets/o1/o1.ob1
```

Creating an object diagram using the graphical interface:

```
use concepts/classes/classes.cl1 concepts/objets/o1/o1.ob1
```

Using the command line interpreter:

```
use -nogui concepts/classes/classes.cl1 concepts/objets/o1/o1.ob1
```

5.1.3 GEdit support

Support for gedit syntax highlighting is **under construction**.

Class model specification:

```
~/.local/share/gtksourceview-3.0/language-specs/
/usr/share/gtksourceview-3.0/language-specs/
```

External tool specification:

```
~/.config/gedit/tools/
```

CHAPTER 6

References

Index

Symbols

A

C

E

G

I

O

P

R

S

T

U