Projets

Consignes Phase 1

- Timing: ~2.5 semaines, à rendre le dimanche 1er Octobre avant minuit en le déposant dans ce répertoire.
- Rendu:
  - Soit un "Cahier des charges", voire-même des "Spécifications", notamment si les contours du projet sont flous;
  - Soit un "Dossier d'analyse fonctionnelle", voire-même un "Rapport de conception détaillée, notamment si le projet est de nature plus technique.
  - Soit un peu des deux, en fonction de ce qui est utile au projet.
  - Selon les méthodes CM Méthodes et CM Documents
  - Voir le Template fourni
  - Voir les Exemples des années passées
- But:
  - Montrer que vous comprenez la première barre du V de la "Méthode en V", et les documents qui lui sont associés.
  - Préparer, concrètement et utilement, le terrain pour la Phase 2:
    - Délimiter les contours du sujet avec le client;
    - Préciser avec lui ses attentes, et ce qui peut effectivement être fait;
    - Comprendre la structure du code pré-existant;
    - Proposer une façon d'y insérer vos solutions techniques;
    - Apprendre les technologies nécessaires.
  - Démarrer enfin l'implémentation.

Consignes Phase 2

- Timing: ~8 semaines, à rendre le mercredi 29 Novembre après-midi.
- Selon la méthode Scrum, cf. CM Méthodes, par itérations de 1 à 2 semaines
- Le rendu:
  - A chaque fin d'itération, les Products & Sprint backlogs, voir Template.
  - A la fin, le projet sous forme de dépôt Github, avec son wiki.
- Le but:
  - Appliquer la "Méthode Scrum".
  - Apprendre à contribuer utilement au développement d'un logiciel de large ampleur. En comprenant l'existant, en concevant / implémentant / testant / documentant du code élégant, stable, compréhensible.

Sujets

- Par équipes de 4, annoncées dans le TableauDeBord.
- Pas deux fois le même sujet, sauf si tous pris.

PyRobots

LaTeXDragon

Secure eXchange Protocols

PyRobots

Les jeux probots, crobots, etc. sont une manière ludique d'apprendre un langage de programmation. Les joueurs y programment, dans ledit langage, le comportement d'un tank. Les différents tanks sont placés sur une arène, et combattent. Le joueur qui a programmé le tank survivant a gagné.
Le langage Python s'impose de plus en plus dans le monde de l'éducation; pour apprendre aux enfants à 'coder', ou encore en Spécialité ISN de Terminale S. Un jeu pyrobots serait donc un excellent outil pédagogique. Malheureusement nous ne connaissons pas de version mature à ce stade. Celle-ci est peu convaincante: [https://bitbucket.org/leeharr/pybotwar].

Nous avons dors et déjà un prototype plus mature, écrit en python dans le framework web Django. Voir :

- Le code et son wiki [https://github.com/huacayacauh/WebPyRobot/wiki]
- Le cahier des charges initial: [https://drive.google.com/open?id=0B2MIU9mHr4WudFE1RGV2TGQwNGs]

A ce stade, plusieurs stratégies sont possibles:

★ *Projet MaturePyRobots.* Il s'agira de prendre contact avec des adolescents et/ou des Profs. de Spécialité ISN de Terminale S et faire évoluer l'existant selon leurs remarques, exigences, perceptions --- jusqu'à parvenir à une version déployable. (Conseil: prendre une présentation dans la catégorie "Python", e.g. Django.)

★ *Projet KivyPyRobots.* Il s'agit de basculer l'existant pour que cela devienne une application mobile/tablette etc. En implémentant de l'auto-complétion de commandes pour que cela reste facile à utiliser malgré l'absence de clavier.
(Conseil: prendre une présentation dans la catégorie "Python", e.g. Kivy, Kivent, PyGame.)

LaTeXDragon

DragonBox [http://www.dragonboxapp.com/video.html] est un jeu éducatif visant à enseigner l'algèbre de façon ludique. Il cible les écoliers et les collégiens. Il s'agit de manipuler des formules via drag'n drop, afin de résoudre des équations.

Nous disposons d'un prototype de version libre du logiciel, avec deux différences majeures. Côté GUI (écrite en JS) nous manipulons directement des expressions LaTeX, rendues en MathJax. Côté backend (écrit en Java) nous sommes très génériques --- l'utilisateur peut changer les règles et décrire par exemple celles de la logique. Voir :

- Le code et son wiki [https://github.com/huacayacauh/LatexDragon/wiki]
- Le cahier des charges initial: [https://drive.google.com/open?id=0BwI3BqLCkW55b3Nvb0lxWTk3bmM]
- La version précédente et son wiki [https://github.com/pja35/Kraken-Free-Dragon/wiki]

A ce stade, plusieurs stratégies sont possibles:

- ★ *Projet MatureDragon.* La version actuelle a quelques défauts, inventoriés notamment ici [https://github.com/huacayacauh/LatexDragon/wiki/Le-futur-de-LatexDragon]. Par exemple la gestion des parenthèses est mal faite: celles-ci sont gérées commes des opérateurs unaires au même titre que '-', alors qu'elles devraient être comprises simplement comme de l'affichage qui apparaît quand on aplatit l'arbre binaire, juste pour désambiguer. Mais il y a d'autres points manquants: par exemple si j'ai *a=b* je peux vouloir ajouter *x* des deux côtés, où *x* est un terme que je saisis au clavier, pour obtenir *a+x=b+x*. De même de drag'n drop est sous-utilisé par rapport au simple clic alors qu'il permet de donner des instructions plus précises; les parties ne sont pas enregistrées côté serveur, etc. Bref, il s'agit de partir de l'existant parvenir à une version complète et déployable.

- ★ *Projet TomDragon.* La version actuelle du jeu est parfois énervante; par exemple pour simplifier *x+y+x* en *2x+y* il nous faut d'abord le réécrire en *x+x+y*, puis rassembler les *x*. Ce serait moins laborieux si nous pouvions faire de la réécriture modulo associativité-commutativité. Or il se trouve que le projet Tom intègre la réécriture d'arbres modulo AC dans Java. D'ailleurs, il a son propre format de descriptions de formules et de règles de réécritures de ces formules. En fait, Tom pourrait avantageusement remplacer tout ou partie du Backend existant. Une première ébauche a été tentée dans cette direction [https://github.com/huacayacauh/LatexDragon/wiki/TOM].

- ★ *Projet JupyterDragon.* Les Computer Algebra Systems comme Matematica, Maple etc. permettent de manipuler des équations sur ordinateur. Sympy est sûrement le plus prometteur des CAS libres (partie intégrante de SciPy, intégrée à la distribution de Python appelée Anaconda). De plus il dispose d'une très belle GUI appelée Jupyter, qui utilise MathJax pour son rendu. Nous souhaitons pouvoir interagir avec Sympy, en drag'n drop, via Jupyter. Par exemple, nous souhaitons pouvoir "tirer *x* en dehors de *3(2x+xy)*" et que cela invoque la fonction méthode *collect(exp,x)* de Sympy, pour nous renvoyer *3x(2+y)*. Une équipe précédente a mis en place un Wiki pour expliquer comment étendre Jupyter [https://github.com/huacayacauh/sageDragon/wiki].
  (Conseil: prendre une présentation dans la catégorie "Python", e.g. Jupyter ou Faster Python.)

Secure eXchange Protocols

Agacée par les *Airbnb*, *Uber*, et autres plateformes centralisées qui prélèvent un pourcentage sur chaque transaction entre petites gens, une communauté de troc hippie-cool a finalement décidé de se prendre en main. Afin d'organiser au mieux ses affaires, voire même de passer à l'échelle, elle s'est mise de mêche avec une communauté type software libre. Cette dernière lui a programmé un client P2P sur lequel chacun peut :

- créer un profil
- publier ses offres
- faire part de ses demandes
- rechercher parmi celles des autres
- ...

Le tout de manière encryptée et signée. Le prototype se dénomme SXP. Il est disponible librement [https://github.com/pja35/SXP/wiki]. Le projet SXP est découpé en 6 modules très indépendants, qui n'interagissent qu'au travers de leurs APIs, chacun doté d'un Wiki. Il bénéficie de >80% de taux de couverture de code par des tests unitaires. La communauté s'est aussi dotée d'un site qui explique la philosophie du projet et ses aspects plus cryptographiques [https://pja35.github.io/SXP/].

A ce stade, plusieurs stratégies sont possibles.

En termes de P2P---module network. C'est probablement le point faible du projet. En effet: la librairie JXTA, qui a eu son heure de gloire, n'est plus maintenue actuellement. Par ailleurs les Internet Service Providers (ISP) cachent trop souvent les adresses IP des machines via NAT, proxys et firewalls---pour être utilisable une librairie P2P doit donc être constamment mise à jour afin de déjouer ces difficultés via uPNP, etc. Pourtant JXTA avait l'avantage de livrer clé en main une "Distributed Hash Table" (DHT), c-à-d une sorte d'index collectif qui permet à tous les pairs de savoir qui d'entre eux détient tel ou tel fichier.

★ *TomSXP*. TomP2P semble être la librairie de P2P la plus maintenue en Java. Elle possède aussi une DHT. Vous ré-implémenterez l'API du module network, en remplaçant JXTA par TomP2P. Vous préserverez les >80% de taux de couverture de code.
(Conseil: prendre la présentation correspondante.)

★ *WebPeerSXP.* Bonne nouvelle pour le monde du P2P: WebRTC est arrivé. Ce standard, implémenté par tous les plus grands Browsers, permet de communiquer directement de Browser à Browser, court-circuitant ainsi toutes les difficultés traditionnelles. Peut-être que SXP, dont la GUI est déjà écrite en JavaScript et exécutée sur un Browser (electron / chrome), pourrait bénéficier de ce canal de communication. Malheureusement cette technologie est relativement récente, et il n'existe pas encore de librairie standard implémentant une DHT via WebRTC. Vous ferez l'état des lieux sur le sujet, des tests, etc. En n'oubliant pas:
    - https://github.com/diasdavid/webrtc-explorer
    - https://github.com/xuset/web-dht
    - https://github.com/nazar-pc/webtorrent-dht
    - OpenPeer.org

- FreedomJS.

Peut-être s'agira-t-il, au final, de contribuer à l'un de ces projets, après les avoir comparés.
(Conseil: prendre les présentations correspondante, sur les DHT et le P2P en JS.)

★ *KadohSXP.* Kademlia est une DHT qui a été implémentée en Javascript pour être exécutée par les Browsers, via Websockets, court-circuitant ainsi certaines des difficultés traditionnelles. Peut-être que SXP, dont la GUI est déjà écrite en JavaScript et exécutée sur un Browser (electron / chrome), pourrait bénéficier de ce canal de communication. Vous ré-implémenterez l'API du module network, en la remplaçant par Kadoh, qui vivera aux côtés de la GUI. Vous préserverez les >80% de taux de couverture de code.
(Conseil: prendre la présentation correspondante.)

En termes de GUI---module client :

★ *Angular2SXP*. La GUI actuelle est développée sous Bootstrap + AngularJS. Angular2 + Typescript proposent un framework modernisé pour les applications JS d'envergure. Après avoir pris connaissance de ces technologies, vous y ré-implémenterez la GUI SXP. Voir le cahier des charges [https://drive.google.com/file/d/0Bypmh2oHgLMkWDN5MzRzRkFSVXM/view?usp=sharing] et une première ébauche [https://github.com/huacayacauh/AngularSXP/wiki], peut-être à revoir. Vous préserverez les >80% de taux de couverture de code.
(Conseil: prendre les présentation sur Angular2.)

★ *ValidationSXP.* Actuellement les données sont validées par le backend et par la GUI. Par exemple lorsqu'on crée un nouveau profil utilisateur, le GUI vérifie le bon format des données, du mail etc. avant de l'envoyer au backend… qui vérifie à son tour. C'est bien que les deux vérifient. Mais il est encore plus important qu'ils le fassent de façon cohérente. Autrement dit, il faut que le Backend donne ses critères de validation des données à la GUI, et qu'elle valide selon ces critères. Si ce n'est pas possible, il faut qu'elle renonce à valider, mais alors il faut faire remonter, depuis le backend vers l'utilisateur, le champs qui est fautif. Bref, il faut rationaliser ça. Attention certains champs sont parfois encryptés, il ne faut pas le juger invalide pour autant. Pour le moment cela entre en conflit avec certaines annotations qui spécifient les formats, dans les entités. Par exemple dans le Backend, dans l'entité Message, le champs messageContent a une taille très grande quand on l'encrypte. Du coup l'annotation @Size est large, bien plus large que celle vérifiée par la GUI. Voir aussi [https://github.com/huacayacauh/ModelSXP/wiki/Travaux-sur-les-validateurs]

En termes de résilience des données--- module résilience :

★ *ResilientSXP*. Actuellement, si je perd mon ordinateur, je perd mon compte, les items que j'ai publié, etc. C'est un désavantage du décentralisé par rapport au centralisé.

Sous Airbnb, si je perd mon ordinateur, je peux en emprunter un autre et me connecter au site, qui aura conservé toutes mes données. Pourtant, rien n'empêche de faire de même en P2P. Au lieu d'être le seul à conserver les données de mon compte utilisateur, pourquoi ne pas les répliquer, encryptées, sur des clients pairs? Vous implémenterez ce mécanisme de résilience. Pour commencer, vous ferez en sorte que chaque entité créée soit envoyée comme message à 5 pairs (si ce n'est pas déjà le cas), qui les conserveront. Ensuite, il faudra réfléchir à ce qu'il se passe en cas d'édition (maintenir une date de dernière modification et arbitrer les versions selon cette date), de suppression (mettre une date de péremption). Il faudra veiller à la maintenance (vérifier que l'entité est encore bien répliquée à chaque refresh). Vous ferez tout cela en respectant le Design Pattern Decorator. Et en préservant les >80% de taux de couverture de code. Voir le cahier des charges [https://drive.google.com/open?id=0B353moHqAnp7QzNRSzNqcHN1S3M] et le wiki [https://github.com/huacayacauh/ResilientSXP/wiki].
(Conseil: prendre une présentation liées au DHT.)

En termes de gestion des tests et des exceptions/warnings/logs---transversal :

★ *LogSXP*. Actuellement SXP affiche beaucoup de messages en console, difficile de faire le tri. Beaucoup d'exceptions sont simplement qualifiées de RunTime. De plus tout ça n'est pas unifié: par exemple jetty lui logue les évènements à sa manière, qui est indépendante du reste. Voir: src/main/java/ressources log4j2 and jettylogin, as well as controller/tools/loggerutilities.
Bref, il faut nous faut un traitement unifié / centralisé de toutes les exceptions / warnings / logs. Les types exceptions doivent être déclarés, les exceptions doivent être priorétisées, informatives, loguées. Autrement dit, tout ce qui se passe doit se trouver dans un unique fichier, et quand il se passe quelque chose, il faut qu'on sache: si c'est important; d'où ça vient; quel utilisateur / session l'a causé; est-ce qu'il réparer.
On pourra aussi songer à afficher certains de ces évènements dans la GUI pour l'utilisateur. Peut-être faut-il mettre en place des Listener JXTA pour récupérer de l'information sur le comportement du P2P.
(Conseil: prendre la présentation liée au Logging)

★ *TestSXP*. Le projet actuel bénéficie de >80% de couverture du code. Mais certaines choses n'ont pas pu être testées. Par exemple nous avons des difficultés à tester le comportement des Bases de données d'une session à l'autre. Voir dans le WIki: Tests/Notes on persistence. Par ailleurs, tous les aspects réseaux ne sont pas testés, voir Wiki : Tests/Notes on network tests. Vous ferez entrer ces deux aspects importants du projet, dans le champs des choses testables par JUnit. Enfin, certains tests ne passent pas lors du gradle build, il faudrait y regarder de plus près.

En termes de signature de contrats--- module protocols :

★ *ContractSXP.* La première étape pour signer un contrat est de se mettre d'accord sur le contrat ! Une première ébauche est en place, cependant celle-ci n'est pas optimale car elle ne permet pas de voir les différences apportées par rapport au contrat proposé initialement. Vous travaillerez donc à faire en sorte que les signataires puissent se mettre d'accord sur un contrat de façon pratique. Ce sujet touche à plusieurs parties du projet : la GUI, la base de donnée, l'échange de messages et la signature de contrat. Tout d'abord, vous implémenterez le système pour deux utilisateurs. Puis, une fois que le système fonctionnera pour 2, vous l'étendrez à un nombre *n* de signataires pour pouvoir faire des contrats plus complexes.
Dans un second temps, une fois les signataires d'accord, on signe le contrat grâce à des Establisher. Cependant, une fois le contrat signé, il faut que l'utilisateur en soit informé : vous allez implémenter le système qui permet de faire remonter l'information depuis le peer jusqu'à la GUI.

★ *BlockChainSXP*. Disons que vous vous êtes mis d'accord sur les objets à échanger avec un autre utilisateur, et aussi sur les termes d'un contrat d'échange. Il faut maintenant signer ce contrat, et ce n'est pas une tâche aisée en P2P! Car on veut que chacun ne puisse pas être dupé par l'autre… La solution envisagée jusqu'ici est l'utilisation de sigma-protocoles, avec les bonnes propriétés décrites ici : https://pja35.github.io/SXP/wiki/SecureContractSigningProtocol/. Nous voudrions utiliser la technologie *blockchain* comme alternative, et nous pensons à *Ethereum* et ses *smartcontracts*. Vous savez quoi ? Des étudiants se sont déjà cassé la tête l'année dernière, leur travail est ici, spécialement pour vous : https://github.com/alex6683/SXP/wiki/. Il s'agit de comprendre leur travail, reproduire leurs tests, terminer les "TODO" du wiki, améliorer le protocole en réfléchissant plus en profondeur aux smartcontracts, et préparer l'intégration du tout à SXP (en particulier, comment rendre plus légère la synchronisation de la blockchain ?). En préservant les >80% de taux de couverture de code. (Conseil: prendre la présentation liée à Ethereum)

★ *SigmaSXP.* Considérons que tout le monde s'est mis d'accord sur les objets à échanger et sur les termes d'un contrat d'échange. Le contrat peut faire intervenir un nombre n de parties (on ne se limite pas à deux). Il faut maintenant signer ce contrat mais ce n'est pas évident en P2P lorsque l'on ne peut se fier à personne ! Une solution est de faire appel à des sigma-protocoles (https://pja35.github.io/SXP/wiki/SecureContractSigningProtocol/) et ceux-ci sont presque entièrement implémentés dans SXP, il ne manque plus qu'une dernière brique. Dans ce type de protocole, il faut toujours avoir une tierce partie (TTP) qui servira d'arbitre en cas de litige. Mais comment choisir ce TTP lorsque l'on ne peut se fier à personne ? L'idée est de tirer une personne au hasard (ce qui revient au même que de tirer un nombre au hasard : https://github.com/pja35/SXP/wiki/Module:-Protocols). Mais comment tirer un nombre au hasard lorsque l'on est plusieurs et en ne faisant confiance à personne ? C'est ce tirage aléatoire que vous allez implémenter. Un état des lieux sur le fonctionnement actuel du processus de choix du TTP ainsi que divers documents utiles au tirage

aléatoire sont présents sur le wiki : https://github.com/pja35/SXP/wiki/Module:-Protocols. Vous pourrez dans un premier temps réfléchir avec seulement deux partis pour passer à un cadre plus général par la suite.
Ce choix nous pousse alors à nous poser d'autres questions :

- Le TTP doit-il nécessairement être un utilisateur (et donc on doit attendre qu'il se connecte pour arbitrer un litige) ou bien le TTP peut être un processus implémenté dans les peers (côté serveur et donc sans avoir vraiment besoin de connexion) ? Placer le TTP du côté peer ne présenterait-il pas un danger de sécurité ?
- Le TTP doit-il nécessairement être centralisé ou peut-on plutôt se reléguer à un groupe de personnes / voisins ? Un document très utile sur ce sujet est : http://www.springerlink.com/content/28b0t21f4e5fhfb9/ (nous demander si vous voulez le pdf).

Une fois l'implémentation du choix du TTP effectuée, vous réfléchirez de manière construite à ces deux questionnements.

QCM Mix

Une personne du département des langues est chargée de mélanger **à la main** des sujets d'examens donnés au format Excel… nous voudrions lui simplifier la vie ! Il s'agit d'implémenter un logiciel pour répondre précisément à son besoin, jusqu'à une mise en place concrète de l'outil sur sa machine (sous windows, il faudra voir l'installation avec son service informatique). Deux tentatives ont échoué les années passées mais laissent de bonnes bases pour mener à bien le projet cette année :

- la première est "autonome" : https://git.framasoft.org/easyProg/QCMix
- la seconde utilise AMC : https://github.com/Snaoui/TER