SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

CPS 2018-2019
Examen Réparti 1: *Lode Runner*
(+ Projet)

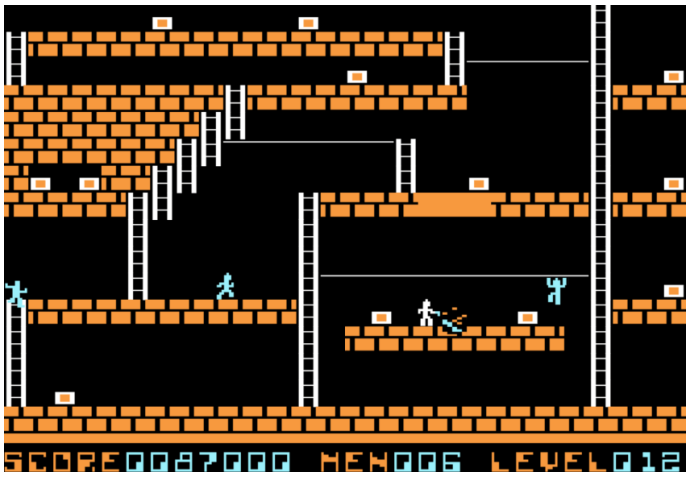

Figure 1: Capture d'écran de *Lode Runner*.

Consignes

- Cet examen évalue la compréhension de la partie *Design-by-contract* du cours et la capacité à produire des spécifications pertinentes. Donner des spécifications complètes et intéressantes de quelques questions sera mieux évalué que traiter les parties "faciles" (liste des observateurs/opérateurs et types) de toutes les questions
- La description donnée dans l'énoncé peut être ambigüe, plusieurs spécifications correctes sont possibles, il faut parfois faire des choix (et expliciter ces choix au maximum dans la copie).
- La spécification doit être semi-formelle. Sa syntaxe peut s'écarter de celle vue en cours et TD si elle est suffisamment claire (quantificateurs $\forall\exists$, ensemble d'éléments, appels d'opérateurs, ...). Des mots-clefs et des prédicats peuvent être définis pour gagner du temps.
- L'examen est plutôt linéaire, et il est conseillé de traiter les exercices dans l'ordre, en laissant possiblement de côté quelques questions.

Description du Jeu Sorti en 1983, *Lode Runner* (Broderbund Software) propose un jeu de plateforme sans défilement (*non-scrolling*) dans lequel le joueur dirige un personnage qui doit récupérer des trésors en évitant des gardes.
Le jeu est vu de côté (une caractéristique du genre *jeu de plateformes*). Sur un écran composé de **plateformes** et d'**échelles**, des **trésors** sont disposés. Le joueur dirige son personnage de manière simple: il peut marcher vers la

gauche ou la droite quand il est sur une plate-forme, et monter, descendre, se laisser tomber à gauche et à droite quand il est sur une échelle. Une fois que tous les trésors de l'écran sont récupérés, une échelle finale apparaît et le personnage peut accéder à l'écran suivant.
Des personnages, controlés par l'ordinateur, appelés **gardes** évoluent dans le même écran que le joueur. Le contact entre un de ses personnages et le joueur se traduit en échec: l'écran se réinitialise et le joueur "perd une vie". Les gardes se dirigent en permanence vers le joueur (c'est donc une des difficultés du jeu que d'éviter les gardes).
En plus de ses actions de mouvement, le joueur dispose de deux actions spéciales: **creuser à gauche** et **creuser à droite** qui lui permettent de creuser un trou dans une plateforme (respectivement à sa gauche et à se droite). Le trou se rebouche tout seul au bout d'un certain temps.
Si un garde tombe dans un trou, il reste coincé un certain temps avant de s'extirper du trou. Le joueur peut marcher par-dessus un garde qui est coincé dans un trou (pour éviter les gardes, le joueur peut ainsi creuser des trous pour piéger les gardes et passer par-dessus eux). Si le trou se rebouche avant que le garde ne s'en extirpe, le garde disparait puis réapparait à sa position initiale.
Certains gardent peuvent porter un trésor. Quand ils tombent dans un trou, il lachent leur trésor directement au dessus d'eux (ce qui permet au joueur de la récupérer en marchant par-dessus eux). S'ils s'extirpent du trou avant que le joueur ne ramasse le trésor, ils le récupèrent.
Les écrans comportent en outre des **rails**, qui agissent comme des échelles horizontales: le joueur et les gardes peuvent s'y suspendre et progresser vers la gauche ou la droite, ou se laisser tomber.
Lode Runner est l'un des premiers jeux intégrant un **éditeur** permettant aux joueurs de construire leurs propres niveaux.

Examen Dans l'examen, on se concentre sur les fonctions de bases du jeu: grille, personnages, et moteur de jeu d'un unique écran (le jeu s'arrête par une victoire quand le joueur récolte tous les trésors).

1 Écran

L'**écran** est l'environnement dans lequel évoluent le joueur et les gardes. Il est *discret*, c'est-à-dire composé de cases (*cells*) et les déplacements se font d'une case à une case adjacente.

La case en bas à gauche de l'écran a les coordonnées $(0, 0)$, celle directement à sa droite les coordonnées $(1, 0)$ et celle directement au dessus d'elle les coordonnées $(0, 1)$.

Une case est soit un vide (*empty cell*), soit une plateforme (*platform*), soit un trou (*hole*), soit une échelle (*ladder*), soit un rail (*handrail*), soit une plateforme en métal (*metal*) (dans laquelle il est impossible de faire un trou). La nature d'une case peut donc être énumérée par le type suivant:

type Cell {**EMP**, **PLT**, **HOL**, **LAD**, **HDR**, **MTL** }

On décrit un service **Screen** pour représenter l'écran. Il possède deux observateurs `Height` et `Width` pour, respectivement, le nombre de lignes (Gauche-Droite) et le nombre de colonnes (Haut-Bas) de l'écran ainsi que `CellNature` qui observe la nature d'une case de l'écran. En outre, un **Screen** possède deux opérations, `Dig` qui transforme une case de nature **PLT** en case de type **HOL** et `Fill` qui fait l'opération inverse.

On suppose qu'à l'initialisation, la nature de toutes les cases d'un Screen est **EMP**.

Q1.1 Compléter la spécification ci-dessous en incluant préconditions, invariants et post-conditions quand c'est nécessaire. **Ne pas recopier** les opérateurs/observateurs et leurs types sur la copie.

Service:	Screen
Observators:	**const** `Height`: [Screen] $\rightarrow$ int
	const `Width`: [Screen] $\rightarrow$ int
	`CellNature`: [Screen] $\times$ int $\times$ int $\rightarrow$ Cell
Constructors:	`init`: int $\times$ int $\rightarrow$ [Screen]
Operators:	`Dig`: [Screen] $\times$ int $\times$ int $\rightarrow$ [Screen]
	`Fill`: [Screen] $\times$ int $\times$ int $\rightarrow$ [Screen]
Observations:	

Q1.2 Donner le code Java de la méthode `init` de la classe `ScreenContract`, implémentation du contrat associé au service **Screen**.

Q1.3 Donner la couverture maximale théorique[1] des **objectifs de préconditions** de la méthode MBT pour le service Screen. Justifier (mais sans détailler les tests).

2 Écran éditable

Le service écran éditable EditableScreen **inclut** le service Screen et y ajoute un nouvel observateur `Playable`, qui indique si un écran est prêt à être joué ou non et une nouvelle opération `SetNature` qui édite directement le contenu d'une case.

Un écran est jouable quand il satisfait aux conditions suivantes:

- Aucune case n'a **HOL** comme nature.
- Toutes les cases de la ligne la plus basse de l'écran ont **MTL** comme nature.

Q2.1 Compléter la spécification ci-dessous avec des préconditions, des invariants et des post-conditions.

Service:	EditableScreen **includes** Screen
Observators:	`Playable`: [EditableScreen] $\rightarrow$ bool
Operators:	`SetNature`: [EditableScreen] $\times$ int $\times$ int $\times$ Cell $\rightarrow$ [EditableScreen]
Observations:	

Q2.2 Cette inclusion est-elle un raffinement ? Justifier.

Q2.3 Donner un cas de test MBT pour l'objectif **transition de `SetNature`**.

Q2.4 Donner un cas de test MBT pour l'objectif **état remarquable: écran jouable** (état dans lequel `Playable` vaut `true`).

3 Personnage et Environnement

Les joueurs et les gardes sont considérés comme des personnages (*characters*). On les décrit au sein du service Character.

L'observateur `Envi` renvoie le Screen dans lequel évolue l'objet. Les observateurs `Hgt` et `Col` renvoient les coordonnées, hauteur (*height*) et colonne (*column*) du personnage.

Un Character est initialisé avec ses coordonnées initiales, un Screen (l'écran dans lequel il évolue). La case initiale du personnage doit être de nature **EMP**. Les Character possèdent quatre opérations de déplacement: `GoLeft` et `GoRight` (pour se déplacer, respectivement vers la gauche ou la droite, quand le personnage est au-dessus d'une plateforme ou accroché à un rail), `GoUp` (pour monter, quand le personnage est sur une case d'échelle), `GoDown` (pour descendre quand le personnage est sur une case d'échelle ou en train de tomber).

Le déplacement se fait selon les règles suivantes:

- Un personnage peut être dans une case seulement si celle-ci est de nature **EMP**, **LAD**, **HDR** ou **HOL** (on utilisera l'adjectif *libre* pour décrire ces types dans la suite).
- Chaque déplacement, quand il est possible, déplace le personnage d'une case uniquement (ainsi, un `GoUp` réussi aura pour effet d'incrémenter l'observateur `Hgt`).
- Un personnage ne peut se déplacer dans une case que si elle est dans les limites de l'écran dans lequel il évolue.
- Un personnage peut monter (opération `GoUp`) seulement si il se trouve dans une case **LAD** et si la case au-dessus de lui est libre.

[1] pourcentage des objectifs atteignables sur l'ensemble des objectifs

- Un personnage peut descendre (opération `GoDown`) seulement si la case au-dessous de lui est libre.

- Un personnage peut aller à gauche (resp. à droite) seulement si la case à sa gauche (resp. à sa droite) est libre et si i) il se trouve au dessus d'une case non-libre ou d'une échelle (**MTL**, **PLT**, **LAD**) ou ii) il se trouve dans une case **LAD** ou **HDR**.

Attention: On fera en sorte qu'on puisse toujours appeler chaque opération de déplacement même quand le déplacement est impossible. Par exemple, si le Character est juste à droite d'une case non-libre (le déplacement à gauche est donc impossible) l'opération `GoLeft` est **possible**, mais n'a pas d'effet (les coordonnées du personnage ne changent pas apres l'opération). Ou encore, si un personnage est sur une case d'échelle et que la case juste au-dessus de lui est une plateforme, l'opération `GoUp` est possible, mais ne change rien.

Q3.1. Donner la spécification de **Character**, avec l'intégralité des signatures, des préconditions et des invariants, ainsi que les postconditions pour `init` et `GoLeft` uniquement.

On veut qu'il y ait au plus un **Character** par case de la grille. Cela implique que les déplacements des différents **Character** ne sont pas indépendants (par exemple, un **Character** ne peut pas se déplacer vers la droite si la case juste à droite est occupée par un autre **Character**).

On décide donc de décrire un sous-service de **Screen** appelé **Environment** qui contient un observateur `CellContent`, qui prend en paramètres des coordonnées et renvoie le contenu de cette case. Cette observateur sera aussi utilisé pour vérifier la présence ou nom d'un objet (comme un trésor) sur cette case.

Pour cela on ajoute un nouveau type de données `Item`:

Data: Item
Observators: `Id`: [Item] $\rightarrow$ int
`Nature`: [Item] $\rightarrow$ ItemType
`Hgt`: [Item] $\rightarrow$ int
`Col`: [Item] $\rightarrow$ int

Pour le moment, le type énumeration ItemType contient uniquement Treasure (le type des trésors).

L'image de `CellContent` est donc le type Set{Character + Item}: le contenu d'une cellule est un ensemble de personnages et de d'objets. Les régles associées à `Environment` sont les suivantes:

- Une case peut contenir au plus un Character.

- Une case non-libre ne contient rien.

- Une case ne peut contenir un trésor que si la case en dessous d'elle est non-libre.

Q3.2. Compléter la spécification de **Environment** ci-dessous.

Q3.3. Peut-on dire que **Environment** raffine **Screen** ? Justifier.

Service: Environment **includes** Screen
Observators : `CellContent`: int $\times$ int $\rightarrow$ Set{Character + Item}

Pour empêcher qu'un **Character** puisse se déplacer dans une case déjà occupée, on modifie l'observateur `Envi` pour qu'il renvoie un **Environment**.

Q3.4. Décrire les changements à apporter à la spécification de **Character** pour que:

- un déplacement (uniquement `GoLeft` dans cet examen) ne soit pas possible dans une case qui contient déjà un personnage (mais toujours possible si la case contient uniquement un trésor).
- un déplacement latéral est possible si la case sous le personnage qui se déplace est libre mais contient un personnage.

4 Gardes

Le service Guard inclue le service Character auquel il ajoute un observateur Id (qui identifie le garde), Target (qui donne la cible du garde, qu'on supposera un personnage), un observateur Behaviour (qui renvoie la prochaine commande que le garde doit effectuer) et deux opérations `ClimbLeft` et `ClimbRight` (qui permettent au garde de sortir d'un trou).

L'image de l'observateur Behaviour est le type énumération Move: {**Right**, **Left**, **Up**, **Down**, **Neutral** }

On donne les règles suivantes:

- Si le garde est dans une échelle, Behaviour renvoie **Up** si la cible du garde se trouve strictement plus haut que lui, **Down** si elle se trouve strictement plus basse, et Neutral sinon.
- Si le garde est dans un trou, sur un rail ou sur une case au-dessus d'une case non-libre ou au-dessus d'une case libre contenant un personnage, Behaviour renvoie **Left** si la cible du garde se trouve strictement plus à gauche que lui, **Right** si elle se trouve strictement plus à droite, et **Neutral** sinon.
- Si le garde est à la fois $i)$ dans une échelle et $ii)$ au-dessus d'une case non-libre ou au-dessus d'une case libre contenant un personnage, le déplacement suggéré par Behaviour suit l'axe (vertical ou horizontal) sur lequel la distance à la cible est la plus courte.
- `ClimbLeft` (resp. `ClimbRight`) n'est possible que si le garde est dans un trou et n'a d'effet que si la case en haut à gauche (resp. en haut à droite) de lui est libre. Si c'est le cas, elle le déplace dans cette case, sinon elle ne fait rien.

Q4.1. Donner la spécification de Guard.

On ajoute maintenant des observateurs et des opérateurs qui permettent de gérer les déplacements au tour-par-tour du garde: TimeInHole observe le temps passé dans un trou par un garde et Step gère l'évolution en un tour de l'état du garde. Ils obéissent aux règle suivantes:

- si le garde se trouve dans une case qui n'est ni une échelle, ni un rail, ni un trou et que la case en dessous de lui est libre, n'est pas une échelle et ne contient pas de personnage, le garde tombe. Dans ce cas l'opération `Step` équivaut à l'opération `GoDown`.
- si le garde se trouve dans un trou et que `TimeInHole` est strictement inférieur à 5[2], l'opération `Step` a pour effet d'incrémenter `TimeInHole`,
- si le garde se trouve dans un trou, que `TimeInHole` vaut 5, et que `Behaviour` vaut **Left** (resp. **Right**) l'opération `Step` équivaut à l'opération `ClimbLeft` (resp. **ClimbDown**). Si `Behaviour` vaut **Neutral** l'opération `Step` n'a pas d'effet observable.
- Sinon le garde obéit à son `Behaviour`, c'est à dire que si `Behaviour(G)` donne **Left**, l'opération `Step` équivaut à l'opération `GoLeft`.

Q4.2. Compléter la spécification de Guard.

5 Joueur

Le service Player inclue le service Character et ajoute un observateur `Engine` et d'une opération Step. L'image de `Engine` est un moteur de jeu (décrit dans la dernière section du sujet) qui possède un observateur `NextCommand` dont l'image est le type énumération Command = Move ∪ {**DigL**, **DigR**}

Q5. Donner la spécification de Player en respectant les règles suivantes pour Step:

[2] valeur arbitraire à régler lors du projet

- Le joueur tombe (comme un garde) quand il ne se trouve pas dans une échelle ou un rail et que la case en dessous de lui est libre et ne contient pas de personnage.
- Sinon le joueur essaye de suivre la commande donnée par `Engine::NextCommand`, c'est à dire de se déplacer à gauche, à droite, en haut ou en bas, ou de creuser un trou à gauche ou à droite quand c'est possible.
- Si `Engine::NextCommand` est **DigL**, que le joueur se trouve au dessus d'une case non-libre ou au dessus d'une case contenant un personnage, que la case à sa gauche (resp. à sa droite) est libre et que la case en bas à gauche (resp. en bas à droite) par rapport à la sienne est de nature **PLT**, cette case est creusée (et devient un trou).

6 Moteur

Le service Engine spécifie le moteur de jeu et doit contenir les observateurs `Environment` (qui observe l'écran courant), `Player` (qui observe le joueur), `Guards` (qui observe l'ensemble des gardes présent dans le jeu), `Treasures` (qui observe l'ensemble des coordonnées des trésors encore présents), `Status` (qui observe l'état du jeu, son image est le type énumération **Playing**, **Win**, **Loss**) et `NextCommand` (qui observe la dernière commande reçue par le l'utilisateur) et un opérateur `Step` (qui avance le jeu d'un tour, en appelant successivement les opérateurs `Step` des gardes et du joueur).

A l'initialisation, on donne un `EditableScreen` jouable, une paire de coordonnées représentant la position initiale du joueur, un ensemble de paires de coordonnées représentant les positions initiales des gardes et un ensemble de paires de coordonnées représentant les positions des trésors (toutes ces positions doivent correspondre à des cases **EMP** différentes de l'écran et les positions des trésors doivent correspondre à des cases situées au dessus d'une case de nature **PLT** ou **MTL**).

On donne les règles suivantes:

- L'observateur `Environment` doit être synchronisé avec les observateurs `Guards`, `Player` et `Treasure`. Ainsi si `G` $\in$ `Guards(E)` est tel que `Hgt(G)`=4 et `Col(G)`=3, alors `G` $\in$ `CellContent(Environment(E),3,4)`.
- Si au début d'un tour, le joueur se trouve sur une case contenant un trésor, ce trésor disparait.
- Le jeu est gagné quand il n'y a plus de trésors.
- Si au début d'un tour, un garde est dans la même case que le joueur, le jeu est perdu (cette règle nécessite de modifier la régle qui interdit à une case de contenir plus d'un personnage, par exemple, en établissant qu'une case ne peut contenir plus d'un **garde**, cette modification n'est pas demandée dans l'examen).

Q6.1 Donner le diagramme de composition de l'état du système apres l'initialisation du service Engine.

Q6.2 Donner la spécification du service Engine

Pour gérer la régénération des trous, on rajoute un observateur `Holes` qui observe un ensemble de triplets (x, y, t) décrivant les trous présents dans l'environnement, avec leurs coordonnées x et y et le temps depuis leur création. On ajoute les règles suivantes:

- L'observateur `Holes` doit être synchronisé avec l'environnement.
- Lorsque le joueur creuse un trou, la troisième coordonnée du trou nouvellement créé t est 0.
- A chaque appel de `Step` du moteur, la troisième coordonnée de chaque trou est incrémentée, puis tous les trous dont la troisième coordonnées vaut 15 sont rebouchés.
- Au moment où un trou est rebouché, si le joueur était dedans, le jeu est perdu.
- Au moment où un trou est rebouché, si un garde était dedans, il revient à sa position initiale.

Q6.3 Donner les modifications à apporter aux différents services pour prendre en compte la gestion des trous.

Sujet de Projet (NE PAS LIRE PENDANT L'EXAMEN)[3]

Le but du projet est de donner une spécification d'un jeu similaire à un *Lode Runner* **amélioré** dont le cahier des charges est partiellement décrit dans le sujet d'Examen Réparti 1, d'implémenter cette spécification selon la méthode *Design-by-Contract* vue en cours, et d'écrire une implémentation pour le jeu.
Le but du projet est de proposer des défis de spécification et d'implémentation de contrats. L'implémentation du jeu a une importance moindre. Toute fonctionnalité implémentée mais non spécifiée a de fortes chances d'être négligée dans l'évaluation.
Le but du projet n'est pas de se rapprocher au maximum de jeux existants. Les initiatives personnelles et originales sont encouragées.

Rendu

Le rendu est composé d'une unique archive contenant:

- la spécification semi-formelle des services utilisés par le projet sous forme d'interface **Java** (e.g. une interface `Player`),
- l'implémentation de spécification par des contrats selon le motif *Decorator* vu en cours (e.g. deux classes `GuardDecorator` et `GuardContract`),
- (au moins) deux implémentations des services spécifiés, une implémentation correcte et une implémentation buggée (e.g. deux classes `GuardImpl` et `GuardImpl`).
- une série de test **JUnit** pertinents, élaborés selon la méthode MBT.
- un fichier `build.xml` permettant la compilation, l'execution et le test des implémentations.
- un rapport en `pdf`.

Etape 1: Implémentation du sujet d'examen

La spécification et l'implémentation des services donnée dans les six exercices de l'examen doivent être entièrement terminée dans le projet.

Etape 2: Complétion de la spécification

Les modifications suivantes devraient être apportées au projet (mais peuvent être oubliées pour d'autres extensions, si nécessaire):

- **Jeu**: Le programme du jeu doit intégrer un éditeur, et permettre le lancement d'une partie et l'intialisation d'un moteur de jeu, qui vérifie des conditions de victoire et de défaite.
- **Gestion des écrans**: Le joueur doit pouvoir charger un écran prédéfini (avec position des personnages et trésors prédéfinies). La collecte de tous les trésors ne doit plus déclencher directement une victoire, mais envoyer le joueur dans un autre écran.
- **Contact**: Le jeu doit pouvoir gérer correctement le contact entre un garde et le joueur, notamment quand un garde s'extirpe d'un trou au-dessus duquel se trouve le joueur.

[3] Sauf si vous avez fini en avance.

- **Vie et Score**: Le jeu doit maintenir un nombre de "vies" pour le joueur, tout échec dans écran décrémente ce compteur et réinitialise l'écran, l'échec d'une partie survient quand ce nombre vaut 0. Chaque trésor ramassé augmente le "score" du joueur (le score gagné dans un niveau réinitialisé suite à un échec est perdu).
- **Gardes**: Certains gardes doivent pouvoir porter un trésor (et le lâcher en tombant dans un trou). L'IA des gardes doit être améliorée par rapport à celle décrite dans le sujet d'examen (par exemple, dans le sujet, si une échelle traverse une plateforme, un garde peut rester coincé sur l'échelle alors que le joueur est sur la plateforme devant lui). Une amélioration incluant une notion de *chemin* (jusqu'au joueur) peut être envisagée.

Etape 3: Extensions

Le jeu *Lode Runner* étant assez simple, pour obtenir une évaluation correcte, le projet doit en plus spécifier et implémenter plusieurs extensions ou modifications (qui doivent être spécifiées pour être prises en compte dans l'évaluation). Le choix des extensions est libre, les idées personnelles d'extension sont encouragées. Voici quelques exemples d'extensions possibles:

- **Combat et Objets**: le joueur peut ramasser des objets (armes ou bonus), combattre (au corps-à-corps ou à distance) les gardes ou affecter l'écran d'autres manières (construire un pont ou une échelle, creuser devant lui).
- **Cases spéciales**: l'écran contient des cases particulières, comme des portes qu'il faut ouvrir avec une clef ou des membranes qu'on ne peut traverser que dans un sens, ou qui ne laissent passer qu'un seul type de personnage, ou des téléporteurs qui transportent le personnage qui passe dans leur case à un autre endroit de l'écran. (Le jeu original comporte aussi des *pièges*: des cases vides qui sont affichées comme étant une case de plate-forme.)
- **Personnages spéciaux**: le jeu gère des personnages spéciaux comme des gardes qui peuvent passer au dessus des trous, ou creuser des murs ou des alliés qui combattent ou retiennent les gardes.
- **Animation de déplacement**: le jeu *Lode Runner* originel, contrairement au jeu spécifié dans le sujet (qui se rapproche de la toute première version proposée par *Douglas Smith*) gère le déplacement *pixel-by-pixel* des personnages (un même personnage peut être à plusieurs endroits différents d'une même case). Cela permet de gérer des "vitesses de déplacement" différentes pour les personnages.
- **Ecran 3D**: le jeu se déroule dans une arène en trois dimensions, comme *Lode Runner II*.

Rapport

Le rapport se compose de 4 parties: un manuel d'utilisation succint de l'implémentation, la spécification formelle complète du projet, la description formelle des tests MBT effectués, et un rapport de projet proprement dit, se concentrant sur les choix et les difficultés de spécification, d'implémentation, de tests, exhibant des exemples choisis pour leur pertinence.

Soutenance

Une soutenance de 15 minutes se compose d'environ: i) 10 minutes de rapport: présentation (rapide) du projet, puis exploration de différents cas de spécification, contrats ou tests pertinents, ii) 5 minutes de démonstration.

Le contenu de la soutenance doit se concentrer sur quelques points précis de spécifications ou de tests. Il est conseillé à chaque groupe de trouver des aspects originaux et spécifiques à présenter et de ne pas chercher à couvrir l'intégralité du projet.