8. Detaillierte Pläne
DeutschOverflow

Supervisor:
Kovács Márton

Members:

Ádám Zsófia	SOSK6A	adamzsofi.mail@gmail.com
Hedrich Ádám	H9HFFV	hedrichadam09@gmail.com
Pintér Balázs	ZGY18G	pinterbalazs21@gmail.com
Fucskár Patrícia	XKYAOO	fucskar.patricia@gmail.com
Tassi Timián	MYY53U	timian.tassi@gmail.com

15. April 2020

8. Detaillierte Pläne

8.1 Pläne von Klassen und Methoden.

8.1.1 Player

- **Verantwortung**
  Diese Klasse ist eine abstrakte Klasse, weil ein Spieler entweder Eskimo oder Forscher ist. Diese Klasse enthält die grundsätzlichen Eigenschaften (Attributen) und die gemeinsamen Methoden beider Arten von Spielern. Natürlich die Eskimos und Forscher Klasse spezialisiert diese Klasse, also enthält weitere Methoden.

- **Schnittstellen**
  - **IControllable**

- **SuperKlassen**
  Character->Player

- **Attributen**
  - **#int Bodyheat:** Die Körpertemperatur des Spielers. Es ist 5 Einheit für Eskimos und 4 Einheit für Forscher.
  - **#int ID:** für eindeutige Charakterisierung des Spielers
  - **#int workPoints:** Der Spieler hat so viele Arbeit (Einheit) noch.
  - **#Item inHand:** Der Gegenstand, der in der Hand des Spielers sich befindet. Es kann 0 oder 1 Gegenstand in der Hand sein. (mehr als 1 nicht)
  - **#DivingSuit wearing:** Es ist eigentlich der Taucheranzug, wenn der Spieler es trägt.
  - **+boolean inWater:** Es zeigt, ob der Spieler im Wasser ist. (true: Ja, false: Nein)

- **Methoden**
  - **+void startRound():** Mit dieser Methode startet ein Spieler ihre Runde.
  - **+void fallInWater():** Der Spieler fällt ins Wasser. Er wartet auf die Rettung.
  - **+void changeBodyHeat(int thisMuch):** Der Spieler kann sich seine Körpertemperatur erhöhen (mit dem Essen) oder es wird durch eine Schneesturm verringert.
  - **+void ateFood():** Der Spieler isst sein Essen, so seine Körpertemperatur wird sich erhöht. *(Bemerkung: dieses Method wird von dem Essen in seinem Hand aufgerufen, weil Essen wird immer automatisch gegessen, falls es aufgenommen ist)*
  - **+void wear(DivingSuit suit):** Der Spieler trägt den Taucheranzug. Wenn er ins Wasser fällt, wird nicht sterben. *(Bemerkung: dieses Method wird von dem Taucheranzug in seiner Hand aufgerufen, weil das Taucheranzug immer automatisch angezogen wird, falls es aufgenommen ist)*
  - **+void pickUp(Item i):** Der Spieler kann damit einen Gegenstand aufnehmen. Wenn in seiner Hand sich schon ein Gegenstand befindet, wird der vorherige Gegenstand abwerfen.
  - **+void clearSnow():** Der Spieler räumt Schnee von der Eisplatte. *(1 oder 2, hängt davon ab, ob das Spieler eine Schaufel in der Hand hat)*
  - **+void digItemUp(Item i):** Der Spieler gräbt ein Gegenstand aus.

- **+void savePlayers(Direction dir):** Ein Spieler kann seinen Kumpel retten. Der Spieler muss eine Richtung eingeben.
- **+void putSignalTogether(SignalFlare sf):** Der Spieler setzt der SignalFackel zusammen.
- **+void passRound():** Der Spieler passt seine Runde.
- **+void dropFragileShovel():** Der Spieler wirft seine zerbrechliche Schaufel ab. *(Wird von zerbrechlicher Schaufel aufgerufen, ähnlich wie beim ateFood/wear)*

8.1.2 Character

- **Verantwortung**
  Diese Klasse ist eine abstrakte "Superklasse". Es ist eigentlich ein Container für alle Charakter, die in dem Spiel vorkommen kann. Es gibt eine gemeinsame Eigenschaft zwischen allen Charakter: der Schritt.

- **Methoden**
  - **+void step(Direction dir):** Diese Methode wird gerufen, wenn ein Charakter treten möchte. Wir müssen die Richtung eingeben.

8.1.3 IControllable

- **Verantwortung**
  IControllable ist eine Schnittstelle. Einige Methoden für die Spieler befinden sich hier. Diese Methoden werden durch Klasse "Player" benutzt: Treten, Gegenstand aufnehmen, schneeräumen, Gegenstand ausgraben, Teile vom Signalfackel zusammenbauen, Runde passen.

- **Methoden**
  - **+ void passRound():** Mit dieser Methode kann der Spieler passen, also das Recht für Tritt weitergeben.
  - **+void digItemUp():** Der Spieler räumt mit Hilfe von einem Gräber Schnee, wenn er ein Gräber hat.
  - **+void pickUp(Item i):** Der Spieler kann ein Gegenstand aufnehmen, wenn der Gegenstand in die Eisplatte nicht einfriert.
  - **+void clearSnow():** Der Spieler macht eine Eisplatte sauber.
  - **+void putSignalTogether(Signalflare sf):** Am Ende des Spiels feuern ein Spieler die Signalfackel. Es ist möglich, wenn alle Spieler in einer Eisplatte steht und die Teile von der Signalfackel da sind. Es kostet 1 Arbeit (Einheit).
  - **+void savePlayers(Direction dir):** Der Spieler rettet sein Kumpel mit Hilfe vom Seil. Man muss eingeben, auf welche Eisplatte (Direction) wird der Spieler gerettet.

8.1.4 Eskimo

- **Verantwortung**
  Diese Klasse spezialisiert die Player Klasse. Es definiert weitere Methoden, weil diese Klasse anders verhalten kann als die andere Spieler Klasse. Den Attributen von dieser Klasse definiert in der Superklasse, weil allen Attributen in beiden Subklassen vorkommen, die Grundwerte können anders sein. (zum Beispiel Körpertemperatur).

- **SuperKlassen**
  Character->Player->Eskimo

- **Methoden**
  - **+void buildIgloo():** Die Eskimos können ein Iglu bauen. Es schützt von dem Schneesturm und von dem Eisbären.

8.1.5 Researcher

- **Verantwortung**
  Diese Klasse spezialisiert die "Spieler" Klasse. Es definiert weitere Methoden, weil diese Klasse anders verhalten kann, als die andere Spieler Klasse. Den Attributen von dieser Klasse definiert in der Vorklasse, weil allen Attributen beiden Subklassen vorkommt, nur die Grundwerte können anders sein. (zum Beispiel Körpertemperatur).

- **SuperKlassen**
  Character->Player->Researcher

- **Methoden**
  - **+void detectCapacity(Direction dir):** Die Forscher können mit dieser Methode anschauen, ob eine Eisplatte stabil oder instabil ist, also wie viel Spieler kann auf eine bestimmten Eisplatte stehen.

8.1.6 PolarBear

- **Verantwortung**
  Diese Klasse verwirklicht der Eisbär. Der Eisbär tritt in jeder Runde in einer zufälligen Richtung. Er kann schwimmen. Diese Klasse vererbt die Superklasse Character.

- **SuperKlassen**
  Character->PolarBear

- **Methoden**
  - +void step(Direction dir): Die "Step" Methode von Character wird überschreibt.

8.1.7 Item

- **Verantwortung**
  Diese Klasse ist eine abstrakte Klasse, also man kann es nicht instanziieren. Es ist ein Container von verschiedenen Gegenständen. Die gemeinsame Eigenschaft von verschiedenen Gegenständen ist, dass es in dem Spiel immer 1 von jedem gibt.

- **Attributen**
  - **+ItemState state:** Es ist der Zustand des Gegenstands. Im Konstruktor wird es am Anfang auf *"frozen"* eingestellt.

- **Methoden**
  - **+void thrownDown():** Der Gegenstand wird abgeworfen. In diesem Fall ist der Gegenstand in die Eisplatte nicht eingefroren und bleibt ebendort.
  - **+void pickedUp(Player picker):** Der Gegenstand wird durch ein bestimmten Spieler aufgenommen. Der Spieler wird im Parameter gegeben.
  - **+void diggedUp():** Mit dieser Methode können wir den Gegenstand ausgraben.
  - **+void used(Player player, Activity activity):** Die verschieden Gegenstände können durch ein bestimmten Spieler benutzt werden. Wir müssen die Aktivität auch eingeben, also was wir mit dem bestimmten Gegenstand machen möchten.
  - **+void ItemState getState():** Es gibt der Zustand des Gegenstands zurück.

8.1.8 Shovel

- **Verantwortung**
  Diese Klasse spezialisiert die "Item" Klasse. Diese Klasse verwirklicht den Schaufel Gegenstand.

- **SuperKlassen**
  Item->Shovel

- **Methoden**
  - **+void used(Player player, Activity activity):** Die verschieden Gegenstände können durch ein bestimmten Spieler benutzt werden. Wir müssen die Aktivität auch eingeben, also was wir mit dem bestimmten Gegenstand machen möchten. (*override)*

8.1.9 Rope

- **Verantwortung**
  Diese Klasse spezialisiert die "Item" Klasse. Diese Klasse verwirklicht den Seil Gegenstand.

- **SuperKlassen**
  Item->Rope

- **Methoden**
  - **+void used(Player player, Activity activity):** Die verschieden Gegenstände können durch ein bestimmten Spieler benutzt werden. Wir müssen die Aktivität auch eingeben, also was wir mit dem bestimmten Gegenstand machen möchten. (*override)*
  - **-Direction getDir():** Wir bekommen eine Richtung, wohin wir den andere Spieler retten möchten.

8.1.10 DivingSuit

- **Verantwortung**
  Diese Klasse spezialisiert die "Item" Klasse. Diese Klasse verwirklicht den Taucheranzug Gegenstand.

- **SuperKlassen**
  Item->DivingSuit

- **Methoden**
  - **+ void pickedUp(Player picker):** Der Gegenstand wird durch ein bestimmten Spieler aufgenommen. Der Spieler wird im Parameter gegeben. (*override)*

8.1.11 Food

- **Verantwortung**
  Diese Klasse spezialisiert die "Item" Klasse. Diese Klasse verwirklicht den Essen Gegenstand. Mit diesem Gegenstand können die Spieler sich ihre Körpertemperatur erhöhen. Falls das Essen gegessen ist, dann es wird auf ein Tile gefroren nochmal erschienen (it "respawns").

- **SuperKlassen**
  Item->Food

- **Methoden**
  - **+ void pickedUp(Player picker):** Der Gegenstand wird durch ein bestimmten Spieler aufgenommen. Der Spieler wird im Parameter gegeben. (*override)*

8.1.12 Tent

- **Verantwortung**
  Diese Klasse verwirklicht das Zelt. Wir haben es als Gegenstand behandelt, aber nur bis es gebaut wird, danach wird es nur als ein "Zähler" arbeiten (nach einem ganzen Runde wird es zerstört).

- **SuperKlassen**
  Item->Tent

- **Attributen**
  - **+int counter:** Es zählt das Leben des Zelts. Wenn es größer als die Anzahl der Spieler, wird der Zelt verschwinden.
  - **+int x,y:** Die Position des Zelt.

- **Methoden**
  - **+void used(Player player, Activity activity):** Die verschiedene Gegenstände können durch ein bestimmten Spieler benutzt werden. Wir müssen die Aktivität auch eingeben, also was wir mit dem bestimmten Gegenstand machen möchten. (*override)*

8.1.13 FragileShovel

- **Verantwortung**
  Diese Klasse verwirklicht die zerbrechliche Schaufel. Es vererbt die Klasse Item.

- **SuperKlassen**
  Item->Shovel->FragileShovel

- **Attributen**
  - **-int counter:** Es zählt das Leben des Schaufels. Wenn es größer oder gleich als 3, wird der Schaufel zerbrechen. *(Es wird 1 größer bei jeder Benutzung)*

- **Methoden**
  - **+void used(Player player, Activity activity):** Die verschiedene Gegenstände können durch ein bestimmten Spieler benutzt werden. Wir müssen die Aktivität auch eingeben, also was wir mit dem bestimmten Gegenstand machen möchten.(*override)*

8.1.14 ItemState

- **Verantwortung**
  ItemState ist ein Enumeration. In dieser Enumeration befinden sich die Zustände von einem Gegenstand. Keine anderen Zustände existieren. In einem Augenblick kann sich zu einem Gegenstand pünktlich ein Zustand gehören.

- **Aufzählungsliterale**
  - **frozen:** Wenn ein Gegenstand gefroren ist, gehört sich zu diesem Zustand.
  - **inHand:** Wenn ein Gegenstand sich in der Hand von einem Spieler befindet, gehört sich zu diesem Zustand.
  - **thrownDown:** Ein Gegenstand kann auf eine Eisplatte nicht in gefroren Zustand sein, also durch einen Spieler abgeworfen wird. In diesem Fall gehört sich zu diesem Zustand

8.1.15 Activity

- **Verantwortung**
  Diese Klasse ist eine Enumeration. Es ist gegeben, welche Aktivitäten ein Spieler durchführen kann. Diese Enumeration speichert diese Möglichkeiten. Andere Aktivitäten können während des Spiels nicht durchgeführt werden.

- **Aufzählungsliterale**
  - **savingPeople:** Ein Spieler kann einen anderen Spieler von dem Wasser retten.
  - **clearingSnow:** Die Spieler können die Eisplatten saubermachen, also der Schnee wird von der Eisplatte weggeputzt.
  - **eatingFood:** Wenn ein Spieler sich mit einem Essen begegnet, isst automatisch es. Danach wird ein anderes Essen irgendwo erschien. Während dieser Tätigkeit wird sich die Körpertemperatur von dem bestimmten Spieler erhöht.
  - **puttingOnSuit:** Der Spieler nimmt den Taucheranzug auf sich. Es wird automatisch durchgeführt, wenn ein Spieler einen Taucheranzug aufnehmen
  - **putUpTent:** Der Spieler kann ein Zelt auf eine Schneeplatte bauen, falls dort sich kein Iglu befindet.

8.1.16 RoundController

- **Verantwortung**
  Diese Klasse ist eine Singleton Klasse. Diese Klasse steuert das Ende und der Beginn eines Spiels, also es ist ein Kontroller. Es beinhaltet ein Spieler Container. In diesem Container befinden sich den Spielern. Die Runden werden auch in dieser Klasse kontrolliert. Am Anfang des Spiels kann man durch diese Klasse den Spielern initialisieren.

- **Attributen**
  - **-int curID:** Es speichert, welche Spieler kommt. Es speichert den Identifikationsnummer des Spielers.
  - **+SignalFlare sg:** Es ist der Signalfackel als Attribut in dieser Klasse.

- **+SnowStorm ss:** Es ist der Schneesturm als Attribut in dieser Klasse.
- **-static RoundController rc:** Globale, Singleton RoundController. Es können wir mit der "*getInstance()"* Methode abfragen.
- **+ArrayList<PolarBear> polarbearList:** Die Eisbäre werden hier gespeichert und kontrolliert. *(Normalerweise haben wir 1 Eisbär, aber die Möglichkeit für mehrere ist mit diesen List gegeben)*
- **+Tent tent:** Der Zustand des Zelts können wir hier kontrollieren. Wenn eine Runde wird ablauft, dann verschwindet das Zelt.

- **Methoden**
  - **+void init(int playerNum):** Mit dieser Methode können wir einen Spieler initialisieren. Dazu müssen wir ein Spieler Nummer eingeben. Diese Nummern identifizieren eindeutig den Spielern, also müssen unterschiedlich sein.
  - **+void startNextRound():** Am Ende der Runde wird die Nächste gerufen. Der erste Spieler der Round setzt das Spiel fort.
  - **+void endLastRound():** Die jetzige Runde beenden.
  - **+void lose(String cause):** Das Spiel beendet. Der Grund der Niederlage wird eingegeben.
  - **+void win():** Die Spielern haben gewonnen, das Spiel wird beendet.
  - **+ int getcurID():** Es gibt den Identifikationsnummer des derzeitigen Spieler zurück.
  - **+ void checkTent():** Nach den Treten des Spieler wird das Lebens des Zelt vermindern. Wenn eine ganze Runde beendet wird, verschwindet das Zelt.
  - **+ static RoundContoller getInstance():** Es gibt der *"rc" zurück.*

8.1.17 SignalFlare

- **Verantwortung**

  Diese Klasse ist eigentlich die Signalfackel. Dieser Gegenstand besteht aus 3 Teile: Pistole, Leuchte, Patron. Mit diesem kann die Spieler das Spiel gewinnen.

- **Attributen**
  - **-ArrayList<SignalFlarePart> signalFlareParts[3]:** Es ist ein Array und dieser Array beinhaltet die 3 Teile vom Signalfackel. Kein Teil des Signalfackels kann verschwinden.

- **Methoden**
  - **+void putTogether(RoundController rc):** Mit dieser Methode können die Spieler das Spiel beenden. Wenn alle Spieler auf derselbe Eisplatte stehen und sie haben alle 3 Signalfackel Teil in der Hand, dann es kostet eine Arbeit diese Methode zu rufen und das Spiel zu beenden.

8.1.18 PlayerContainer

- **Verantwortung**
  Diese Klasse beinhaltet die Spieler. Die Spieler werden durch ihren Identifikationsnummer eindeutig identifiziert.

- **Attributen**
  - **-int playerNum:** Die Anzahl des Spielers.
  - **-static PlayerContainer pc:** Es ist ein Instanze vom Playercontainer. Es ist globale und es können wir mit der "*getInstance()"* Methode abfragen.
  - **-ArrayList<Player> players:** Diese Liste beinhaltet die Spieler in dem Spiel.

- **Methoden**
  - **+ static PlayerContainer getInstance():** Es gibt der *"pc" zurück.*
  - **+Player getPlayer(int pid):** Diese Methode gibt ein Spieler zurück. Wir müssen den Identifikationsnummer dem bestimmten Spieler eingeben. Die verschiedenen Nummern werden zu verschiedenen Spielern zugeordnet.
  - **+static void Initialize(int num):** Wir können am Anfang einen neue Playercontainer initialisieren.
  - **+int getPlayerNum():** Es gibt "*playerNum"* zurück.

8.1.19 SnowStorm

- **Verantwortung**
  Diese Klasse erzeugt den Schneesturm. Dieser Sturm verringert die Körpertemperatur des Spielers, fall er sich nicht in einem Iglu oder in einer Zelt und erhöht sich die Dicke der Schneeschichten auf die verschiedenen Eisplatten.

- **Methoden**
  - **+ void tryStorm():** Diese Methode erzeugt eigentlich den Schneesturm.

8.1.20 SignalFlarePart

- **Verantwortung**
  Diese Klasse speichert ein Teil von Signalfackel. Es gibt 3 Teile und das Ziel des Spiels ist, diese 3 einzubauen und abzuschießen. Die verschiedenen Teile sind verschiedene Instanzen von dieser Klasse

- **SuperKlassen**
  Item->SignalFlarePart

- **Attributen**
  - **-int partID:** Die Identifikationsnummer von dem Teil. Es muss eindeutig sein.

- **Methoden**
  - **+void used(Player player, Activity activity):** Die verschieden Gegenstände können durch ein bestimmten Spieler benutzt werden. Wir müssen die Aktivität auch eingeben, also was wir mit dem bestimmten Gegenstand machen möchten.

8.1.21 PositionLut

- **Verantwortung**

  Diese Klasse ist eine Singleton Klasse. Diese Klasse handelt sich die Bewegungen von den Spielern. Den Spielern können mit Hilfe von dieser Klasse von einer Platte auf eine andere Platte treten. Des Weiteren können wir die Position von verschiedenen Spielern abfragen oder welchen Gegenständen befindet sich in einer bestimmten Eisplatte.

- **Attributen**
  - **#static PositionLUT pLUT:** Unser einzige *"PositionLUT"* Instanze.
  - **-HashMap<Item, Tile> itemTileMap:** Hier wird die Eisplatte-Gegenstand Paare gespeichert.
  - **-HashMap<Tile,ArrayList<Item>> tileItemMap:** Zu der Eisplatte gehörende Liste des Gegenstandes.
  - **-HashMap<Player, Tile>playerTileMap:** Hier wird der Spieler-Gegenstand Paare gespeichert.
  - **-HashMap<PolarBear, Tile> polarbearTileMap:** Hier wird zu dem Eisbären eine Eisplatte geordet. (Wo er sich befindet.)
  - **-HashMap<Tile,ArrayList<PolarBear>> tilePolarBearMap:** Zu der Eisplatte gehörende Liste des Eisbären. *(Jetzt haben wir nur ein Eisbär, aber wir haben die Möglichkeit, mehrere zu haben.)*
  - **-ArrayList<ArrayList<Tile>> tileList:** Die Liste der Eisplatten.
  - **-HashMap<Tile,ArrayList<Player>> tilePlayerMap:** Zu der Eisplatte gehörende Liste des Spielers.

- **Methoden**
  - **+Tile getPosition(Player p):** Mit dieser Methode können wir die Position des bestimmten Spieler abfragen. Wir müssen einen Spieler eingeben und es wird zurückgegeben, auf welche Eisplatte steht er.
  - **+Tile getPosition(Item i):** Mit dieser Methode können wir die Position des bestimmten Gegenstand abfragen. Wir müssen einen Gegenstand eingeben und es wird zurückgegeben, auf welche Eisplatte befindet sich es.
  - **+ArrayList<Player> getPlayersOnTile(Tile t):** Diese Methode ergibt eine Spieler ArrayList. In dieser ArrayList befindet sich ein Spieler, falls er auf die eingegebenen Platte steht, also müssen wir eine Eisplatte eingeben.
  - **+ArrayList<Item> getItemOnTile(Tile t):** Diese Methode ergibt eine Gegenstand ArrayList. In dieser ArrayList befindet sich ein Gegenstand, falls es auf die eingegebenen Platte steht, also müssen wir eine Eisplatte eingeben.
  - **+Tile getTile(int x, int y):** Es gibt eine Eisplatte anhand der Koordinaten zurück.
  - **+void setPosition(Player p, Tile t):** Die Position des Spielers wird auf eine Eisplatte eingestellt.
  - **+void setPosition(Player p, Item i):** Die Position des Gegenstandes wird auf eine Eisplatte eingestellt.

- **+Tile getPosition(PolarBear pb):** Es gibt die Position des Eisbärs zurück.
- **+void setPosition(PolarBear pb, Tile t):** Die Position des Eisbärs wird auf eine Eisplatte eingestellt.

8.1.22 Tile

- **Verantwortung**
  Diese Klasse ist ein Container, also zusammenfasst alle Art von den Eisplatten. Die Klasse ist abstrakt, man kann es nicht instanziieren. Es gibt 3 Arten von den Eisplatten. Es kann stabil, instabil oder ein schneebedecktes Loch sein. Diese Arten sind verschiedene Subklassen, vererben diese abstrakten Klasse.

- **Attributen**
  - **#int x:** Es ist die x Koordinate (horizontale) von der bestimmten Eisplatte.
  - **#int y:** Es ist die y Koordinate (vertikale) von der bestimmten Eisplatte.
  - **#int snow:** Diese Attribute speichert, wie viel Schnee (wie viel Einheit) sich auf die Eisplatte befindet.
  - **#boolean iglooOn:** Diese Attribute bestimmt, ob ein Iglu sich auf diese Eisplatte befindet. (true: Ja, false: Nein)
  - **+boolean tentOn:** Es zeigt, ob ein Zelt sich auf diese Eisplatte befindet.
  - #**int capacity:** Die Kapazität der Eisplatte.
  - **+int standingHere:** Die Anzahl der Spieler, die auf diese Platte stehen.

- **Methoden**
  - **+void steppedOn(Player p):** Ein Spieler tritt auf die Eisplatte. Wir müssen den Spieler eingeben.
  - **+void steppedOff(Direction dir):** Ein Spieler tritt auf eine andere Eisplatte. Wir müssen die Richtung eingeben. Direction ist eine Enumeration, also von bestimmten Richtungen (4 Himmelsrichtung) können wir wählen.
  - **+void changeSnow(signed int thisMuch):** Die Höhe des Schneeschicht verändert sich. Es wird eingegeben, mit wie vielen. Es kann höher oder kleiner sein. Zum Beispiel höher, falls ein Schneesturm kommt.
  - **+Tile getNeighbour(Direction dir):** Mit dieser Methode können wir die benachbarten Eisplatten von einer Platte abfragen. Wir müssen die Richtung auswählen und eingeben.
  - **+void destroyIgloo():** Diese Methode baut den Iglu, der auf diese Eisplatte steht, ab.
  - **+void buildIgloo():** Eskimos können auf eine bestimmten Platte ein Iglu bauen.
  - **+int getSnow():** Diese Methode gibt die Schneedicke der Eisplatte zurück.
  - **+int getCapacity():** Diese Methode gibt die Kapazität der Eisplatte zurück.
  - **+boolean getIglooOn():** Diese Methode gibt den Wert des "*iglooOn"* Attribut zurück.
  - **+void putOnTent():** Mit dieser Methode kann der Spieler ein Zelt auf die Eisplatte bauen.
  - **+boolean equals(Tile t):** Es liefert zurück, ob die Position von zwei Eisplatten gleich ist.

8.1.23 SnowyHole

- **Verantwortung**
  Diese Klasse spezialisiert die "Tile" Klasse. Diese Klasse verwirklicht das schneebedeckte Loch. Diese Eisplatten siehe so aus, wie die andere schneebedecktes Platte, aber wenn ein Spieler auf diese tritt, fällt ins Wasser. Auf diesen Platten können 0 Spieler stehen.

- **Superklasse**
  Tile-> SnowHole

- **Methoden**
  - **+void steppedOn(Player p):** Ein Spieler tritt auf die Eisplatte. Wir müssen den Spieler eingeben. (*override*)
  - **+void destroyIgloo():** Diese Methode baut den Iglu, der auf diese Eisplatte steht, ab. (*overrirde*)
  - **+void buildIgloo():** Eskimos können auf eine bestimmten Platte ein Iglu bauen. (*override)*

8.1.24 StableTile

- **Verantwortung**
  Diese Klasse spezialisiert die "Tile" Klasse. Diese Klasse verwirklicht die stabile Eisplatte. Auf diesen Platten können unendlich viel Spieler stehen. Auf diesen Platten kann sich natürlich auch Schnee befinden.

- **Superklasse**
  Tile-> SnowHole

8.1.25 UnstableTile

- **Verantwortung**

Diese Klasse spezialisiert die "Tile" Klasse. Diese Klasse verwirklicht die instabile Eisplatte. Auf diesen Platten können nur begrenzt viel Spieler stehen. Die Anzahl den Spielern ist nicht sichtbar, aber die Forscher können es anschauen.

- **Superklasse**
  Tile-> SnowHole

- **Methoden**
  - **+void steppedOn(Player p):** Ein Spieler tritt auf die Eisplatte. Wir müssen den Spieler eingeben. (*override*)
  - **+void steppedOff(Direction dir):** Ein Spieler tritt auf eine andere Eisplatte. Wir müssen die Richtung eingeben. Direction ist eine Enumeration, also von bestimmten Richtungen (4 Himmelsrichtung) können wir wählen. (*override*)

8.1.26 Direction

- **Verantwortung**

Diese Klasse ist eine Enumeration und beinhaltet die verschiedenen Richtungen. Wenn ein Spieler treten möchte, sollte er von diesen Richtungen wählen. Es ist eigentlich die 4 Himmelrichtungen. Andere Richtung existiert in diesem Spiel nicht, also können die Spieler diagonale weise nicht treten.

- **Aufzählungsliterale**
  - **up:** Auf die nördliche Platte treten. (Nord)
  - **down:** Auf die untere Platte treten. (Süd)
  - **left:** Auf die linke Platte treten. (West)
  - **right:** Auf die rechte Platte treten. (Ost)
  - **here:** Der Spieler bleibt auf die Platte.

- **Attributen**
  - **-int value:** Zu der bestimmten Richtung gehörende Wert. (zwischen 0 und 4)
  - **-static HashMap<int,Direction> map:** Es speichert der Wert-Direction Paare.

- **Methoden**
  - **+static Direction valueOf(int direction):** Es ist eine int->Direction Umwandlung.
  - **+static int valueOf(Direction dir):** Es ist eine Direction->int Umwandlung.
  - **+int getValue():** Es gibt den "*value*" Attribut zurück.

8.2 Detaillierte Pläne von Testfällen, Beschreibungen

Wir testen das Programm mit dem deterministischen Map, so brauchen wir PrintMap (Character, Item) Kommanden nicht unbedingt, es kann aber manchmal hilfreich sein. Wir können Printkommanden auch mit watch Funktion von Debugger ersetzen, das ist aber nicht unbedingt bequemer.

8.2.1 Used test(Food)

- **Beschreibung**

Beobachtung von Temperaturänderung, wenn ein Spieler ein Essen (Food) aufnimmt.

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Testet used Funktion von Food und pickedUp Funktion von Player

o **Eingänge**	o **Erwartete Ausgänge**
PrintPlayer //Körpertemperatur check	Player Eigenschaften //Beobachtung, jetzt Temperatur ist wichtig für uns
PrintCharacterMap //sehen wo unsere Spieler steht	PlayerMap //Beobachtung ItemMap //Beobachtung
PrintItemMap //ItemCheck	„Picked up Food" //erwartete, falls ein Food und aktivPlayer auf einem Eisplatte sind, sonst:
PickUp //Aktion, Food addiert sofort 1 Temperatur	„Nothing to pick up"// „Picked up <other Item>"
PrintItemMap //sehen ob das Essen verschwindet	ItemMap //Food musste verschwinden
PrintPlayer //Körpertemperatur check	Player Eigenschaften // Temperatur muss mit 1 grösser sein, falls „Picked up Food" ablauft

8.2.2 Used test(DivingSuit)

- **Beschreibung**

Wir nehmen ein Player und ein DivingSuit auf derselben Eisplatte, und sehen wir, dass das Divingsuit in Wearing geht.

- **Getestete Funktionalitäten, erwartbare Fehler**

Test von used Funktion von DivingSuit, wear und pickUp Method von Player.

○ **Eingänge**	○ **Erwartete Ausgänge**
PrintPlayer //Beobachtung	Player Eigenschaften //Beobachtung, jetzt Temperatur ist wichtig für uns
PrintCharacterMap //Beobachtung	PlayerMap //Beobachtung
PrintItemMap //Beobachtung	ItemMap //Beobachtung
PickUp //Aktion	„Picked up DivingSuit" //erwartete, falls ein DivingSuit und Player auf einem Eisplatte sind
PrintItemMap //DivingSuit verschwindet	ItemMap //DivingSuit musste verschwinden
PrintPlayer //Wearing attribut check	Player Eigenschaften //Wearing attribut muss DivingSuit sein.

8.2.3 Used test(Rope)

- **Beschreibung**

Wir nehmen ein Player mit ein Rope und mindestens ein Player in einem benachbarten Feld, in dem Wasser. Wir testen hier die Rettung.

- **Getestete Funktionalitäten, erwartbare Fehlern**

Hier testen wir savePlayer Method von Player, used Method von Rope, step Method von Player kann man auch hier testen, ob es gut funktioniert.

○ **Eingänge**	○ **Erwartete Ausgänge**
PrintCharacterMap//Beobachtung	PlayerMap
SavePlayers <dir>//Richtung wählen	„Saved Players <Player IDs>//falls möglich sonst no rope oder no player to save Warnungen
Step <dir>//Spielern in Wasser können treten (falls mehrere Spieler, dann mehrere Step)	"Step successful, stepped from (x,y) to (x,y)"//falls Spielern treten auf gute Feld, sonst: „step not succesful" oder „Player fell in water" //letzte wird auf deterministische Map nicht möglich.
PrintCharacterMap//War die Rettung erfolgreich?	PlayerMap//check, Spielern sind nicht auf dem WasserFeld.

8.2.4 Used test(Shovel)

- **Beschreibung**

Wir nehmen ein Player mit einer Schaufel. Wir testen hier Schneeräumung mit Schaufel. Bemerkung: Wir können eine zerbrechliche Schaufel auch testen: wenn wir es dreimal rufen, und am Ende sehen wir das inHand mit PrintPlayer. Wenn es leer ist, dann funktioniert die zerbrechliche Schaufel gut.

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Testet clearSnow Method von Player, used Method von Schaufel.

Eingänge	**Erwartete Ausgänge**
PrintSnowMap// Beobachtung	SnowMap
PrintCharactersMap//Beobachtung	CharacterMap
PrintPlayer //InHand Shovel/fragileShovel?	Player Eigenschaften
ClearSnow//Aktion	“Cleared <1 oder 2> Snow off from Tile. ”//clear confirm
PrintSnowMap// snow auf Tile muss mit 2 weniger sein (kann nicht negativ sein, min = 0)	SnowMap//snow prüfen auf Tile, wo Player das Schnee räumt. es muss 2 weniger sein (>=0)
Optional für zerbrechliches Schaufel: nach 3 erfolgreiche Clearsnow: PrintPlayer, InHand muss leer sein.	Optional für FragileShovel: Player Eigenschaften nach 3 „Cleared”, InHand muss leer sein

8.2.5 Used test(Tent)

- **Beschreibung**

Wir nehmen ein Player mit einem Zelt in Hand, und sehen wir was passiert, wenn er es benutzt.

- **Getestete Funktionalitäten, erwartbare Fehlern**

Testet used Method von Zelt.

Eingänge	**Erwartete Ausgänge**
PrintHeimMap//map von Iglus und Zelt	HeimMap
BuildTent//Aktion	"Succesfully built a tent"// kann „Can't build tent here" oder „I have no tent in my hand" sein
PrintHeimMap//sehen ob Zelt platziert ist oder nicht	HeimMap// falls erfolgreich: mit neuem Zeltoptional: wenn wir nach eine Rund es sehen, muss das Zelt verschwinden

8.2.6 PutTogether Test

- **Beschreibung**

Wir sehen in ItemMap mit PrintItemMap, und PlayerMap mit PrintCharacterMap. Wenn alle sind auf einem feld, und alle SignalFlarePart ist dort, dann sollte auf dem Bildschirm: "The signal flare is done!" Sonst entweder: "We don't have all the 3 parts" oder "All players should stand here to do that"

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Testet putTogether Methode von Signalfackel.

Eingänge	**Erwartete Ausgänge**
PrintCharacterMap//alle Player muss auf einem Feld sein	CharacterMap//alle Player muss auf einem Feld sein
PrintItemMap//hier müssen wir kein SignalFlarePart sehen→ die sind inHand	ItemMap//hier müssen wir kein SignalFlarePart sehen→ die sind inHand
PutSignalTogether//Aktion	"The signal flare is done!"//wenn Bedingungen sind gut, sonst kann: "We don't have all the 3 parts" /"All players should stand here to do that"

8.2.7 Player Test

- **Beschreibung**

Test von oben noch nicht getesteten Methoden in Player Klasse.

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Es testet startRound, fallInWater, digItemUp und passRound Methoden von Player, andere Methoden haben wir früher getestet in used-testen.

o **Eingänge**	o **Erwartete Ausgänge**
PassRound// 2mal Aktion	"Player <ID> passed"//wenn es erschient zweimal, mit verschiedene ID-s, dann wissen wir, dass aktuelle Player verändert. Es testet passRound und startRound auch.
PrintItemMap//Beobachtung(Item&Zustand)	ItemMap//Items mit Zuständen
DigItemUp//mit einem Player über ein Item mit frozen itemState.	„digged up <Item>"// diggedUp test. Könnte auch „no frozen Item here" sein.
PrintItemMap//Zustandsänderungs-check	ItemMap//digged up Item muss in thrownDown Zustand sein.
PrintCharacterMap	CharacterMap// wo ist aktuelle Player
Step < dir : in Wasser >//fallInWaterTest, step ist schon getestet	"Player fell in water"//fallInWater Methode geruft
PrintPlayer	Player Eigenschaften//inWater ==true

8.2.8 Researcher Test

- **Beschreibung**

Wir testen mit einem Researcher, ob detectCapacity Method gut funktioniert. Es kann man mit der Vergleichung der erwarteten und wirklichen Ausgänge machen.

- **Getestete Funktionalitäten, erwartbaren Fehlern**

DetectCapacity Method von Researcher wird getestet.

o **Eingänge**	o **Erwartete Ausgänge**
UseSkill <dir>//sehen das capacity won gegebene Tile	"Capacity of Tile (x, y) is <Capacity>" //number, kann „no tile there" sein
PrintTile <x,y>//Vergleichung, x,y sind von dir und playerpos kalkuliert	Tile Eigenschaften//capacity muss gleich sein, falls Tile existiert

8.2.9 Eskimo Test

- **Beschreibung**

Wir testen mit einem Eskimo, ob buildIgloo Method gut funktioniert. Es kann man auch mit der Vergleichung der erwarteten und wirklichen Ausgänge.

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Testet buildIgloo Method.

- **Eingänge**
- **Erwartete Ausgänge**

PrintHeimMap //Beobachtung	HeimMap//Beobachtung
UseSkill //Eskimo baut ein Iglu	“Succesfully built an igloo”// “Can’t build igloo here”: wenn ein Iglu schon dort war
PrintHeimMap //check	HeimMap //ein neuer Iglu muss erschienen.

8.2.10 TileTest (Auch für abgeleitete Klassen: StableTile, UnstableTile, SnowyHole)

- **Beschreibung**

Dieser Test ist in 8.2.1-8.2.9 Testfallen schon gemacht, wir können diese Klassen „direkt” mit Kommanden nicht erreichen, aber andere Methoden benutzen es und der fehlerfreie Ablauf von diesen Methoden sind benötigt.

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Wir haben getNeighbour() Methode, und die steppedOff und steppedOn Funktionen von abgeleiteten Klassen (StableTile, UnstableTile, SnowyHole) getestet. In getNeighbour Funktion soll man auf Indexierung und gute Rückgabewert besonders achten

8.2.11 RoundController Test

- **Beschreibung**

Dieser Test ist schon fast gemacht, wenn man das Spiel benutzt und es scheint gut laufend, dann ist es ok. (Dieser Test ist basiert auf anderen Testen die auch gut laufen)

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Hier Testen wir nur lose Methode.

Eingänge	**Erwartete Ausgänge**
Step <in water dir>//ein Spieler geht ins Wasser	"Player fell in water"
PassRound//jeder Spieler passt	"Player <ID> passed"//Anzahl: SpielerAnzahl – 1
	„lose <cause>"//in diesem Fall <cause> == „drowning", kann aber „polarBearAttack" oder „cold" wenn wir das deterministisch proto Map benutzen, dann können wir beide testen.

8.2.12 SnowStorm Test

- **Beschreibung**

In Proto-Version wurde auch ein deterministisches Drehbuch dafür angezeigt, um zu testen und demonstrieren.

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Es testet die Wirkung von Schneesturm.

Eingänge	**Erwartete Ausgänge**
PrintHeimMap	HeimMap//Beobachtung
PrintSnowTileMap//Beobachtung	SnowTileMap// Beobachtung
PrintPlayer//Player muss auf einem Feld sein wo Schneesturm kommt (deterministisch!)	Player Eigenschaften//Temperatur ist interessant
	SnowTileMap// nach SnowStorm muss in einige Felden snow++
PrintSnowTileMap//nach einer Rund, wenn ein Schneesturm kommen muss.	
PrintPlayer//Falls kein Iglu oder Zelt: Temperatur--, sonst bleibt.	Player Eigenschaften// Falls kein Iglu oder Zelt: temperatur--, sonst bleibt.
PrintHeimMap	HeimMap//Zelt, Iglus mussen zerstört werden, wo Storm war

8.2.13 PolarBear Test

- **Beschreibung**

In proto-Version gibt es eine deterministische Version, das testen wir auch hier.

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Wenn ein PolarBear und ein Spieler treffen, das Spiel beendet. Es ist in endLastRound von RoundController gemacht.

- **Eingänge**

Step dir<polarbear nextPos dir>

- **Erwartete Ausgänge**

"Step successful, stepped from (x,y) to (x,y)"//polarBear ist deterministisch, wir wissen wohin es geht

„lose <cause>"// cause: PolarBear

8.2.14 JUnit test: State check(Item)

- **Beschreibung**

Wir nehmen ein Item, rufen wir die Methoden, die zu Zustandsänderungen führen (thrownDown, pickedUp, diggedUp), und testen, ob der Zustand der Gegenstand (ItemState) des Items gut ist nach diesen Methoden.

- **Getestete Funktionalitäten, erwartbare Fehlern**

Zustandsänderungen sind hier getestet, wir beobachten itemState Membervariable von einem Item. Mögliche Fehler kann zB.: falsche Zustandsänderung sein.

- **Eingänge**

In JUnit wir testen thrownDown, pickedUp, diggedUp Methoden von einem Item.

- **Erwartete Ausgänge**

Nach thrownDown(): von InHand :itemState == ThrownDown

Nach diggedUp(): von frozen :itemState == ThrownDown

Nach pickedUp(): von ThrownDown :itemState == InHand

8.2.15 JUnit test: PlayerContainer Test

- **Beschreibung**

Unit-test von getPlayer Method in PlayerContainer Klasse. PlayerContainer ist deterministisch initialisiert.

- **Getestete Funktionalitäten, erwartbaren Fehlern**

Hier vergleichen wir das Rückgabewert mit players Arraylist Elementen.

- **Eingänge**

Wir geben verschiede Nummern als getPlayer Parameter, und vergleichen wir das Rückgabewert mit players ArrayList.

- **Erwartete Ausgänge**

Gleichheit ist für gute Ergebnis.

8.3 Spezifikation der Test unterstützenden Hilfs- und Übersetzungsprogramme

Wir werden JUnit benutzen um Unit-testen zu machen. JUnit ist ein open source Java Testing Framework. Es hat built-in support in Intellij IDEA. Mit JUnit können wir testen das erwartete Ergebnis von einem Method mit konkreten Parametern gut ist, wir können auch erwartete Ausnahmen (Exceptions) prüfen.