#pragma once

#include "AbstractToken.h"

// The base class from which all Pieces (i.e. occupants of Decks and Maps) shall be derived.

class AbstractPiece {

public:

// Returns true iff this AbstractPiece has been completely tokenized.

virtual bool isTokenized() const = 0;

// Iteratively returns the tokens that constitute this AbstractPiece. Throws an exception if

// this Piece has already been tokenized.

virtual AbstractToken\* tokenize() = 0;

// Writes this Piece to the standard output stream.

virtual void display() const = 0;

};

#pragma once

// The base class from which all Tokens (i.e. occupants of TokenGraphs) shall be derived.

class AbstractToken {

public:

// Returns true iff the specified Tokens have the same type.

static bool areSameType(AbstractToken\*, AbstractToken\*);

// Constructs a new AbstractToken object.

AbstractToken();

// Constructs a new AbstractToken object with the same placed status as the specified Token.

AbstractToken(const AbstractToken&);

// Destroys this AbstractToken.

~AbstractToken();

// Returns true iff this Token occupies a TokenGraph.

bool isPlaced() const;

// Returns the type of this Token.

virtual int getType() const = 0;

// Sets the placement status of this Token to placed.

void place();

// Writees this Token to the standard output stream.

virtual void display() const = 0;

private:

bool\* placed;

};

#pragma once

#include <iostream>

// Compares the expected expression to the the actual expression, and prints the appropriate

// message given the result. Note that equality is predicated on identity semantics unless the

// equality operator is overriden for the type of the expected and actual expressions.

#define ASSERT\_EQUALS(expected, actual, success, failure) { \

if((expected) == (actual)) \

std::cout << success << std::endl; \

else \

std::cerr << failure << std::endl; \

}

// Evaluates the specified expression and prints the specified message if successful.

#define ASSERT\_SUCCESS(expression, msg) { \

try { \

(expression); \

std::cout << msg << std::endl; \

} catch (...) { \

std::cerr << "ERROR:: exception thrown"; \

} \

}

// Evaluates the specified expression and attempts to catch the specified exception. Should the

// expression evaluate successfully, or if an exception other than the one specified is thrown, the

// specified message is printed.

#define ASSERT\_THROWS(exception, expression, msg) { \

try { \

(expression); \

std::cerr << msg << std::endl; \

} catch (exception& e) { \

std::cout << "Successfully caught excption: "; \

std::cout << e.what() << std::endl; \

} catch (...) { \

std::cerr << msg << std::endl; \

} \

}

#pragma once

#include <exception>

#include "AbstractPiece.h"

#include "BuildingToken.h"

using BuildingType = BuildingToken::BuildingType;

// The Piece that belong to BuildingDecks and that occupy VGMaps.

class Building : public AbstractPiece {

friend class VGMapLoader;

public:

// Constructs a new Building object.

Building();

// Constructs a new Building object with the specifeid type and value. Throws an exception if

// the specified value is not between one and the Height of a VGMap.

Building(BuildingType, int);

// Constructs a new Building object with the same attributes as the specified Building.

Building(const Building&);

// Destroys this Building.

~Building();

// Returns the type of this Building.

int getType() const;

// Returns true if Building is Face Up

bool isFaceUp() const;

// Returns the value of this Building.

int getValue() const;

// Returns true iff this Building has been completely tokenized.

bool isTokenized() const;

// Returns the BuildingToken that constitutes this Building. Throws an exception if this

// Building has already been tokenized.

BuildingToken\* tokenize();

// Flips this Building. If it was previously face up, it becomes face down and vice versa.

// Throws an exception if this BuildingToken has already been placed

void flip();

// Writes this Building to the standard output stream.

void display() const;

private:

BuildingToken\* token;

};

#pragma once

#include "AbstractToken.h"

// The token type that inhabits spaces on a VGMap.

class BuildingToken : public AbstractToken {

public:

// The types a BuildingToken may have. (Corresponding ResourceTypes has corresponding

// ordinals.)

enum class BuildingType {

FOREST,

MEADOW,

QUARRY,

WHEATFIELD

};

// Constructs a new BuildingToken object with a random type.

BuildingToken();

// Constructs a new BuildingToken object with the specified type and value. Throws an exception

// if the specified value is not between one and the height of a VGMap.

BuildingToken(BuildingType, int);

// Constructs a new BuildingToken object witht same type and value as specified BuildingToken.

BuildingToken(const BuildingToken&);

// Destroys this BuildingToken.

~BuildingToken();

// Returns true iff this BuildingToken is face up.

bool isFaceUp() const;

// Returns the type of this BuildingToken.

int getType() const;

// Returns the value of this BuildingToken.

int getValue() const;

// Flips this BuildingToken. If it was previously face up, it becomes face down and vice versa.

// Throws an exception if this BuildingToken has already been placed

void flip();

// Writes this BuildingToken to the standard output stream.

void display() const;

private:

BuildingType\* type;

int\* value;

bool\* faceUp;

static BuildingType randomType();

static int randomValue();

static int validateValue(int);

};

#pragma once

#include <vector>

#include "../maps/TokenGraph.h"

#include "../pieces/HarvestTile.h"

#include "../ScoringFacilities.h"

// The common board onto which HarvestTiles are placed.

class GBMap {

static constexpr int DEFAULT\_NUM\_PLAYERS = 2;

static constexpr int DIM\_MIN = 10, DIM\_MAX = 14;

public:

// Constructs a new GBMap object.

GBMap();

// Constructs a new GBMap object for the specified number of players. Throws an exception if

// the specified number of players is not between two and four inclusive.

GBMap(int);

// Constructs a new GBMap object with the same configuration as the specified GBMap.

GBMap(const GBMap&);

// Destroys this GBMap.

~GBMap();

// Places the specified HarvestTile onto the specified square of this GBMap. Throws an

// exception if the specified HarvestTile is null or has already been placed; or if the

// specified square is not on this GBMap or if it is already occupied.

void setSquare(HarvestTile\*, std::pair<int, int>);

// Counts, starting from the specified square, and records on the specified GatherFacility the

// number of connected resources of each type on this GBMap. Throws an exceotion if the

// specified square is not on this GBMap.

void calculateResources(std::pair<int, int>, GatherFacility\*);

// Writes this GBMap to the standard output stream.

void display() const;

// Returns the number of connected nodes on the graph underlying the specified GBMap.

friend int numberOfNodes(GBMap&);

private:

int\* numPlayers;

TokenGraph\* graph;

static std::vector<std::pair<int, int>> expand(std::pair<int, int>);

void setNumPlayers(int);

int height() const;

int width() const;

std::vector<std::pair<int, int>> coordinatesOf(std::pair<int, int>, bool = false);

void validateSquare(std::pair<int, int>);

bool isOnCorner(int, int);

};

#pragma once

#include "../maps/GBMap.h"

#include "../util/Scanner.h"

// A utility to read a serialized GBMap and return it as an object.

class GBMapLoader {

public:

// Supress default constructor.

GBMapLoader() = delete;

// Constructs a new GBMapLoader from the specified path. Throws an exception if the file at the

// specified path cannot be opened.

GBMapLoader(const std::string&);

// Suppress copy constructor.

GBMapLoader(const GBMapLoader&) = delete;

// Destroys this GBMapLoader.

~GBMapLoader();

// Returns a GBMap read from the file specified at the creation of this GBMapLoader. Throws an

// exception if the specified file is not properly formatted or if it contains invalid map data.

GBMap\* load();

private:

Scanner\* scanner;

int getNumPlayers();

HarvestTile\* nextTile();

ResourceToken\* nextToken();

std::pair<int, int> nextSquare();

std::string errorMessage(const std::string&);

};

#pragma once

#include <vector>

#include "AbstractPiece.h"

#include "ResourceToken.h"

// The Piece that occupies HarvestTileDecks and GBMaps.

class HarvestTile : public AbstractPiece {

friend class GBMapLoader;

public:

// The number of resources that a HarvestTile holds.

constexpr static int NUM\_RESOURCES = 4; // TODO this is not actually a constant

// Constructs a new HarvestTile object.

HarvestTile();

// Constructs a new HarvestTile object with the same resources and configuration as the

// specified HarvestTile.

HarvestTile(const HarvestTile&);

// Destroys this HarvestTile.

~HarvestTile();

// Rotates this HarvestTile 90 degrees clockwise the specified number of rotations. Throws an

// exception if this HarvestTile has already been tokenized.

void rotate(int);

// Returns true iff this HarvestTile has been completely tokenized.

bool isTokenized() const;

// Iteratively returns the ResourceTokens that constitute this HarvestTile. Throws an

// exception if this HarvestTile has already been tokenized.

ResourceToken\* tokenize();

// Writes this HarvestTile to the standard output stream.

void display() const;

private:

int\* current;

std::vector<ResourceToken\*>\* resources;

// For GBMapLoader.

HarvestTile(int);

int validateRotation(int);

};

class Player {

public:

Player();

// Constructs a new Player object with the same attributes as the specified Player.

Player(const Player&);

// Destroys this Player.

~Player();

// Returns this Player's village board.

VGMap\* getVillageBoard() const;

// Induces this Player to draw a Building from the specified Deck. Throws an exception if the

// specified Deck is null or empty.

void drawBuilding(BuildingDeck\*);

// Induces this Player to draw a HarvestTile from the specified Deck. Throws an exception if

// the specified Deck is null or empty.

void drawHarvestTile(HarvestTileDeck\*);

// Induces this Player to build the selected Building on their VGMap at the specified circle.

// Throws an exception if any of the following conditions hold:

// (1) the selected building is not consistent with this Player's hand,

// (2) the specified circle is not on this Player's VGMap or it is already occupied,

// (3) specified Building is not being placed adjacent to another of the same type

// (4) the specified Building is being played face up on a circle that is not the correct value.

void buildVillage(int, std::pair<int, int>);

// Adjusts the specified GatherFacility's count of the specified type according to

// specified cost. Throws an exception if the specified type is out of the specified

// GatherFacility's bounds, if the specified GatherFacility is null, or if the

// resources available to this Player are insufficiet to cover the specified cost.

void resourceTracker(GatherFacility\*, int, int);

// Counts the villagers attracted to this Player's village.

void calculateScore();

// Induces this Player to attempt to place the selected HarvestTile onto the specified GBMap at

// the specified square. Throws an exception if the specified tile selection is not consistent

// with this Player's Hand, if the specified GBMap is null, or if the specified square is not

// on the GBMap this Player is playing on.

void placeHarvestTile(int, GBMap\*, std::pair<int, int>);

// Counts the resources available to this Player ont he specified GBMap, starting from the

// specified square, and records the results on the specified GatherFacility

// exception if the specified GBMap or GatherFacility is null, or if the specified

// not on the GBMap this Player is playing on.

void calculateResources(GBMap\*, std::pair<int, int>, GatherFacility\*);

// TODO const, rename, and comment these

void printHarvestTileHand();

void printBuildingHand();

private:

HarvestTileHand\* tiles;

BuildingHand\* buildings;

VGMap\* villageBoard;

BuildFacility\* buildFacility;

#pragma once

#include <vector>

#include "pieces/AbstractPiece.h"

#include "pieces/Building.h"

#include "pieces/HarvestTile.h"

// The base class from which all Decks (i.e. containers for Pieces) shall be derived.

class Deck {

public:

// Constructs a new Deck object.

Deck();

// Destroys this Deck.

~Deck();

// Returns true iff this Deck has no pieces in it.

bool isEmpty() const;

// Returns the number of Pieces in this Deck.

int getSize() const;

// Removes and returns the Piece at the top of this Deck.

virtual AbstractPiece\* draw() = 0;

protected:

// Puts the specified Piece on the top of this Deck.

void push(AbstractPiece\*);

// Removes the Piece at the top of this Deck.

AbstractPiece\* pop();

// Returns a list view of this Pieces in this Deck.

std::vector<AbstractPiece\*>\* asList() const;

private:

std::vector<AbstractPiece\*>\* pieces;

};

// The Deck that contains Buildings.

class BuildingDeck : public Deck {

public:

// Constructs a new BuildingDeck object.

BuildingDeck();

// Constructs a new BuildingDeck object that contains copies of the Buildings contained by the

// specified BuildingDeck.

BuildingDeck(const BuildingDeck&);

// Adds the specified Building to this BuildingDeck. Throws an exception if the specified

// Building is null.

void add(Building\*);

// Removes and returns the Building at the top of this BuildingDeck. Throws an exception if

// this BuildingDeck is empty.

Building\* draw();

class HarvestTileDeck : public Deck {

public:

HarvestTileDeck();

// Constructs a new HarvestTileDeck object that contains copies of the HarvestTiles contained

// by the specified HarvestTileDeck.

HarvestTileDeck(const HarvestTileDeck&);

// Adds the specified HarvestTile to this HarvestTileDeck. Throws an exception if

// the specified HavestTile is null.

void add(HarvestTile\*);

// Removes and returns the HarvestTile at the top of this HarvestTileDeck. Throws

// an exception if this HarvestTileDeck is empty.

HarvestTile\* draw();

class HarvestTileHand {

public:

// Constructs a new HarvestTileHand object.

HarvestTileHand();

// Contstruts a new HarvestTileHand object with the specified shipment tile.

HarvestTileHand(HarvestTile\*);

// Constructs a new HarvestTileHand object with same contents as specified Hand.

HarvestTileHand(const HarvestTileHand&);

// Destroys this HarvestTileHand.

~HarvestTileHand();

// Adds the specified HarvestTile to this HarvestTileHand.

void insert(HarvestTile\*);

// Returns the HarvestTile selected by a Player. Throws an exception if the

// specified selection is not between one and two inclusive.

HarvestTile\* exchange(int);

// Return this HarvestTileHand's shipment tile. Throws an exception if this Hand

// does not contain a shipment tile.

HarvestTile\* ship();

// prints all the harvest tiles contained in the hand

void display();

private:

HarvestTile\* one;

HarvestTile\* two;

HarvestTile\* shipment;

bool isEmpty();

bool isFull();

};

class BuildingHand {

public:

// Constructs a new BuildingHand object.

BuildingHand();

// Constructs new BuildingHand object with same contents as BuildingHand.

BuildingHand(const BuildingHand&);

// Destroys this BuildingHand.

~BuildingHand();

// Adds the specified Building to this BuildingHand.

void insert(Building\*);

// Returns Building selected by Player. Throws an exception if specified selection

// does not exist.

Building\* select(int);

// prints all the buildings contained in the hand

void display();

private:

std::vector<Building\*>\* ownedBuildings;

bool isEmpty();

#pragma once

#include "AbstractToken.h"

// The token type that comprises HarvestTiles.

class ResourceToken : public AbstractToken {

public:

// The types a ResourceToken may have. (Corresponding BuildingTypes have corresponding

// ordinals.)

enum class ResourceType {

SHEEP,

STONE,

TIMBER,

WHEAT

};

// Constructs a new ResourceToken object with a random type.

ResourceToken();

// Constructs a new ResourceToken object with the specified type.

ResourceToken(ResourceType);

// Constructs a new ResourceToken object with the same type as the specified ResourceToken.

ResourceToken(const ResourceToken&);

// Destroys this ResourceToken object.

~ResourceToken();

// Returns the type of this ResourceToken.

int getType() const;

// Writes this ResourceToken to the standard output stream

void display() const;

private:

ResourceType\* type;

static ResourceToken::ResourceType randomType();

};

#pragma once

#include <fstream>

// A utility to read the contents of a file.

class Scanner {

public:

// Suppress default constructor.

Scanner() = delete;

// Constructs a new Scanner object from the specified file path. Throws an exception if the

// file at the specified path cannot be opened.

Scanner(const std::string&);

// Suppress copy constructor.

Scanner(Scanner&) = delete;

// Destroys this Scanner.

~Scanner();

// Returns the line number of the file specified at the creation of this Scanner that it is

// currently at.

int line() const;

// Returns the column number of the file specified at the creation of this Scanner that is is

// Currently at.

int column() const;

// Returns true iff the file specified at the creation of this Scanner has not been entirely

// consumed.

bool hasNext();

// Reads the next character of the file specified at the creation of this Scanner and matches

// it against the specified character. Throws an exception initialized with the specified

// message if the characters do not match.

void consume(char, const std::string&);

// Returns the next character of the file specified at the ceration of this Scanner. Throws an

// exception if the file has been consumed.

char nextChar();

// Returns the next character of the file specified at the creation of this Scanner,

// interpreted as a boolean value. Throws an exception if the file has been consumed.

bool nextBool();

// Reads the next sequence of bytes of the file specified at the creation of this Scanner that

// can be interpreted as an integer, and returns them as such. Throws an exception if the file

// has been consumed or the next character of the file cannot be interpreted as an integer.

int nextInt();

private:

int\* \_line;

int\* \_col;

std::streampos\* start;

std::ifstream\* stream;

char advance();

bool isDigit(char);

};

#pragma once

#include <map>

// Records the number of colonists attracted to a Player's village.

class BuildFacility {

public:

// Constructs a new VillageBuildingScoringFacility object.

BuildFacility();

// Constructs a new VillageBuildingScoringFacility obeject with the same score as the specified

// VillageBuildingScoringFacility.

BuildFacility(const BuildFacility&);

// Destrorys this VillageBuildingScoringFacility

~BuildFacility();

// Returns the score recorded by this VillageBuildingScoreFacility.

int getScore() const;

// Increments the score recorded by this VillageBuildingScoreFacility by the specified amount.

void incrementBy(int);

private:

int\* score;

};

// Records the number of resources available to Players.

class GatherFacility {

public:

// Constructs a new GatherFacility object.

GatherFacility();

// Constructs a new GatherFacility object initialized to the same state as the specified

// GatherFacility.

GatherFacility(const GatherFacility&);

// Destroys this GatherFacility.

~GatherFacility();

// Returns the amount of the sepcified resource type that has been recorded on this

// GatherFacility. Throws an exception if the specified type is not between 0 and the number of

// resource types.

int countOf(int) const;

// Adjusts the amount of the specified resource type recorded on this GatherFacility by the

// specified amount. Throws an exception if the specified type is not between 0 and the number

// of resource types.

void incrementBy(int, int);

// Writes this GatherFacility to the statndard output stream.

void displayCount() const;

private:

// key: ResourceType value: amount

std::map<int, int>\* count;

int validateType(int) const;

};

#pragma once

#include <bitset>

#include <map>

#include <set>

#include "../pieces/AbstractToken.h"

// A graph that holds AbstractTokens.

class TokenGraph {

public:

static constexpr int NUM\_TYPES = 4;

// Static factory method: returns a TokenGraph configured as a grid of the specified height

// and width.

static TokenGraph\* gridOf(int, int);

// Constructs a new TokenGraph object.

TokenGraph();

// Suppress copy constructor.

TokenGraph(const TokenGraph&) = delete;

// Destroys this TokenGraph.

~TokenGraph();

// Adds a node to this TokenGraph at the specified coordinate.

void addNode(std::pair<int, int>);

// Adds an edge incident on the Nodes at the specified coordinates of this TokenGraph. Throws

// an exception if either of the specified coordinates does not reference a Node of this

// TokenGraph.

void addEdge(std::pair<int, int>, std::pair<int, int>);

// Returns the Token occupying this TokenGraph at the specified coordinate, or null if the Node

// is empty. Throws an exceptionif the specified coordinate does not referecnce a Node of this

// TokenGraph.

AbstractToken\* tokenAt(std::pair<int, int>) const;

// Returns true iff this TokenGraph contains a Token of the specified type.

bool hasType(int) const;

// Returns true iff any of the Nodes of this TokenGraoh that are adjacent to the one at the

// specified coordinate contain a Token of the specified type. Throws an exception if the

// specified coordinate does not reference a Node of this TokenGraph.

bool adjacentHolds(std::pair<int, int>, int) const;

// Places the specified Token into the Node at the specified coordinate of this TokenGraph. The

// status of the specified Token will be placed after calls to this method return. Throws an

// exception if the specified coordinate does not reference a Node of this TokenGraph.

void setTokenAt(AbstractToken\*, std::pair<int, int>);

// Initiates a breadth-first search of this TokenGraph from the Node at specified coordinate.

// Returns the number of Nodes that are connected to the source Node, which contain the same

// type of Token as source. Throws an exception if the specified coordinate does not reference a

// Node of this TokenGraph.

int search(std::pair<int, int>);

// Returns true iff the Node at the specified coordinate of this TokenGraph has been previously

// reached from another by means of a call to search(). Throws an exception if the specified

// coordinate does not reference a Node of this TokenGraph.

bool isSearched(std::pair<int, int>);

// Puts the Nodes of this TokenGraph into a native state (i.e. the one that they are in before

// any search has been performed).

void cleanupSearch();

// Returns a mapping from the coordinates of this TokenGraph to the Tokens at those

// coordinates.

std::map<std::pair<int, int>, AbstractToken\*> tokens();

private:

class Node {

public:

// White: requires processing, Gray: in processing, Black: processed, Red: do not process

static constexpr int WHITE = 0, GRAY = 1, BLACK = 2, RED = 3;

AbstractToken\* token;

std::set<Node\*>\* adjacents;

// Search Attribute.

int\* color;

Node();

Node(const Node&) = delete;

~Node();

// Initializes this Node with the specified Token and adjacency set. Should the type of

// specified Token match that of the other, the Node that carries it will be reachable

// from a search initiated from a Node carrying a Token of that type.

void init(AbstractToken\*, AbstractToken\*, std::set<Node\*>\*);

};

std::map<std::pair<int, int>, Node\*>\* nodes;

std::bitset<NUM\_TYPES>\* types;

Node\* nodeAt(std::pair<int, int>) const;

std::pair<int, int> validateCoordinate(std::pair<int, int>) const;

int search(Node\*);

void setupSearchAttributes(AbstractToken\*);

};

#pragma once

#include <array>

#include "../pieces/Building.h"

#include "../ScoringFacilities.h"

#include "TokenGraph.h"

// The Board on which Players build their village.

class VGMap {

public:

static constexpr int HEIGHT = 6, WIDTH = 5;

// Constructs a new VGMap object.

VGMap();

// Constructs a new VGMap object with the same configuration as the specified VGMap.

VGMap(const VGMap&);

// Destroys this VGMap.

~VGMap();

// Places the specified Building onto the specified circle of this VGMap. Throws an exception

// if any of the following conditions hold:

// (1) the specified Building is null,

// (2) the specified circle is not on this VGMap or it is already occupied,

// (3) the specified Building is not being placed adjacent to another of the same type, or

// (4) the specified Building is being played face up on a circle that is not the correct value.

void setCircle(Building\*, std::pair<int, int>);

// Counts the villagers that have been attracted to this VGMap. Throws an exeption if the

// specified BuildFacility is null.

void calculateScore(BuildFacility\*);

// Writes this VGMap to the standard output stream.

void display() const;

private:

TokenGraph\* graph;

static bool valuesMatch(const Building\*, int);

void validatePlacement(const Building\*, std::pair<int,int>);

int countRows();

int countCols();

};

#pragma once

#include "../maps/VGMap.h"

#include "../util/Scanner.h"

// A utility to read a serialized VGMap and return it as an object.

class VGMapLoader {

public:

// Suppress default constructor.

VGMapLoader() = delete;

// Constructs a new VGMapLoader from the specified path. Throws an exception if the file at the

// specified path cannot be opened.

VGMapLoader(const std::string&);

// Suppress copy constructor.

VGMapLoader(VGMapLoader&) = delete;

// Destroys this VGMapLoader.

~VGMapLoader();

// Returns a VGMap read from the file specified at the creation of this GBMapLoader. Throws an

// exception if the specified file is not properly formatted or if it contains invalid map data.

VGMap\* load();

private:

Scanner\* scanner;

Building\* nextBuilding();

BuildingToken\* nextToken();

std::pair<int, int> nextCircle();

std::string errorMessage(const std::string&);

};