Задание на УВП

Разработайте и напишите, используя инструментарий Qt, компьютерную игру типа Adventure (квест).

Что такое Adventure Games?

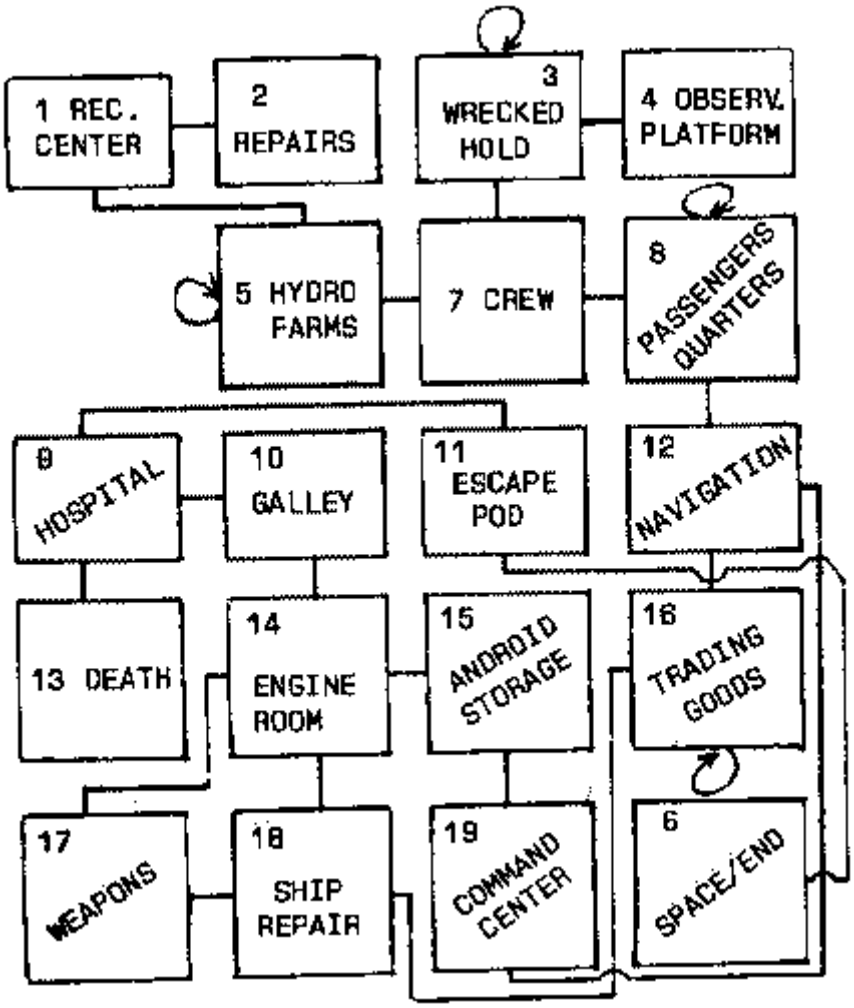
В самой простой версии – это интерактивная игра, где герой, управляемый игроком, попадает в какое-либо помещение (замок, космический корабль и т.д.). Каждая комната имеет описание, и, переходя из одной в другую, герой должен выполнить задание – например, найти какой-то предмет, спасти принцессу и пр. Но загвоздка заключается в том, что заранее не известна «планировка» места, как располагаются комнаты, в какой из них живут голодные драконы, а где спрятано золото. И по ходу игры у игрока вырисовывается карта местности с сопутствующей информацией.

Рекомендуемый порядок выполнения задания.

1. ***Продумайте основные моменты игры:***

   а) определите место действия и задачу игрока;

   б) разработайте карту местности, например карта для космического корабля может выглядеть следующим образом:



   Для простоты каждая комната может иметь не более 4х выходов. Можно делать многоэтажные лабиринты, добавив специальные комнаты-лестницы, лифты или телепортеры, пневмолифты и т.д.;

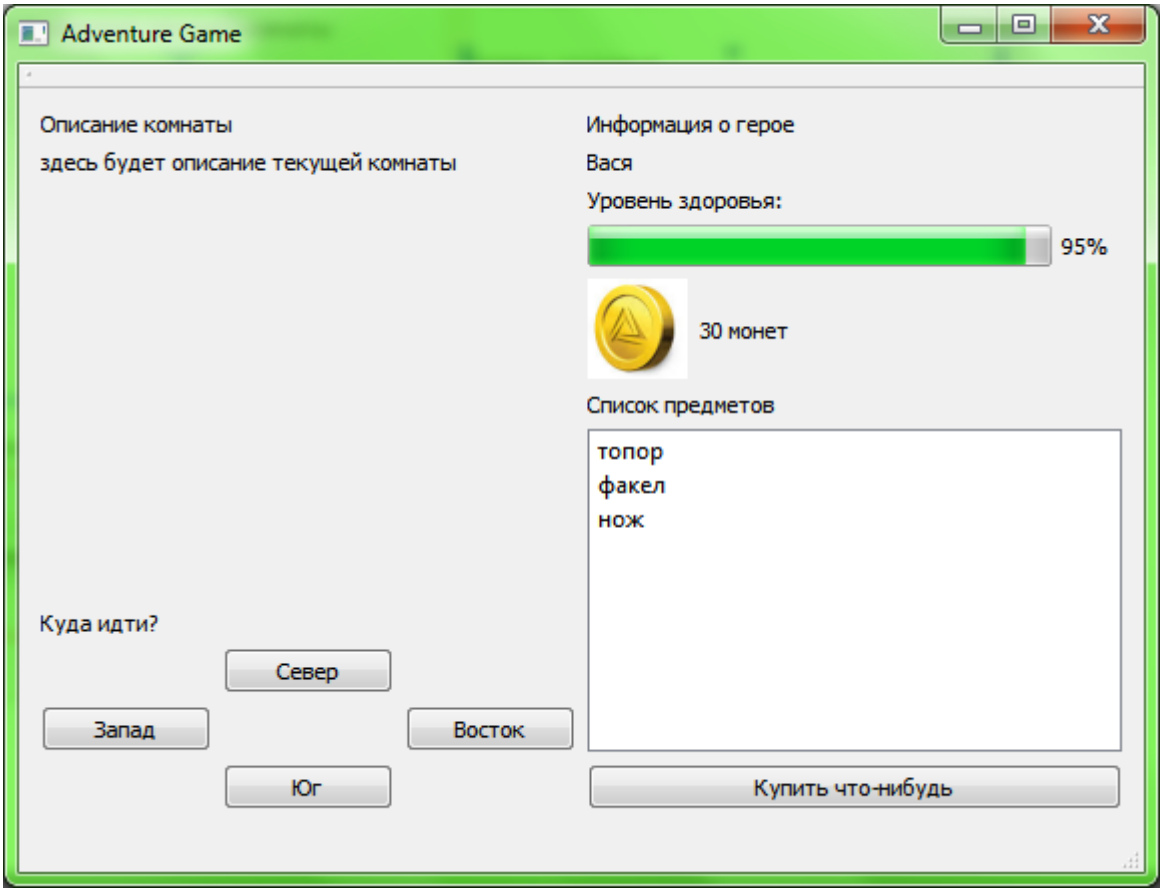
в) придумайте, какие враги будут мешать герою. Наделите их разными свойствами: имя, сила (параметры для случайного распределения), уровень ущерба в случае поражения в бою с ними;
г) продумайте оружие и другие предметы, которые могут находиться в комнатах и как их может использовать игрок;
д) свойства, описывающие игрока (уровень жизни, сила, количество денег).

Также можно добавить:
-- комнаты, в которых герою темно и обязательно зажигать факел (а его нужно либо найти, либо купить);
-- можно ограничить максимальное число предметов, которые герой может носить с собой;
-- в некоторые из комнат добавить степень их «вредности» -- у героя уменьшается здоровье или он погибает;
-- и т.д.

2. *Разработайте интерфейс игры*

В QtCreator создайте проект (File->New File or Project->Application->Qt Widgets Application).
В полученном проекте в папке Forms откройте файл с расширением .ui.
Разместите необходимые для игры виджеты на форме.

Один из вариантов внешнего вида главного окна:



3. Разработайте базовую версию модели для игры

1. Создайте перечисление Directions для хранения возможных направлений движения:
   `enum class Directions {North=0, East, South, West};`

2. Создайте структуру данных для хранения комнат и связей между ними.
   Информацию об одной комнате можно представить с помощью структуры или класса следующего вида:

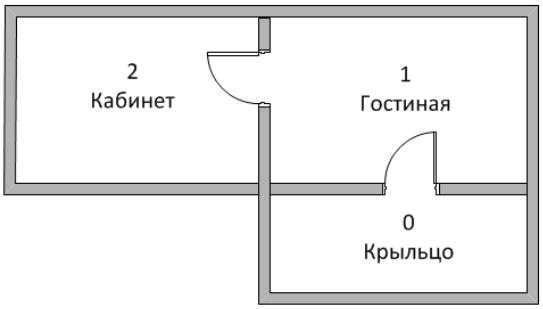
```
class Room
{
public:
     QString name;           // Название комнаты.
     QString description;    // Описание комнаты.

     QMap<Direction,int> neighbourRooms; // Номера соседних комнат,
                                 // индексируются с помощью Direction.

 Room(QString roomName, QString roomDescr, int n, int e, int s, int w)
 :name(roomName), description(roomDescr)
  {
    neighbourRooms[Direction::North]=n;
    neighbourRooms[Direction::East]=e;
    neighbourRooms[Direction::South]=s;
    neighbourRooms[Direction::West]=w;
  }
};
```

   Для хранения списка соседних комнат используется отображение из Direction в int, реализованное с помощью шаблонного класса QMap. Это отображение ведёт себя как массив, индексируемый типом Direction. Значениями этого отображения являются целые числа, задающие номера комнат.

   В проекте класс Room может включать больше полей в зависимости от замысла игры.

3. Создайте класс Maze для хранения лабиринта.
   Для хранения комнат можно использовать список (например, QList<Room> rooms), который будет являться членом этого класса. Реализуйте в классе Maze оператор [] для доступа к элементам этого списка по индексу.

   Пока конфигурация комнат будет жестко задана в коде программы. Создайте в классе Maze метод init() для инициализации игры. Далее показан небольшой фрагмент карты и соответствующая функция инициализации.



```
void Maze::Init()
{
 rooms.append(Room(“Крыльцо”, ”Вы стоите на крыльце. Перед Вами
        находится совершенно загадочная дверь.”, 1, -1, -1, -1 ));
 rooms.append(Room(“Гостиная”, ”Вы попали в гостиную. В камине сидит
        страшный жирный паук.”, -1, -1, 0, 2));
 rooms.append(Room(“Кабинет”, ”Вы стоите в кабинете. Книжный шкаф полон
        старинных книг.”, -1, 1, -1, -1));
}
```

Номера комнат являются порядковыми номерами записей в списке. Значение -1 означает, что в стене нет двери.

Вызовите метод init() в конструкторе класса Maze.

4. Теперь создайте класс Hero для игрока. Этот класс будет посылать сигналы, поэтому он должен быть унаследован от QObject и содержать строку с текстом Q_OBJECT в начале объявления класса. Чтобы автоматически сгенерировать этот код воспользуйтесь средствами QtCreator создания класса и укажите, что ваш класс унаследован от QObject.

   Игрок должен хранить информацию о том, где он находится. Для этого в классе Hero должны быть предусмотрены поля Maze *maze (указатель на текущий лабиринт) и int currentRoom (номер комнаты, в которой находится игрок).

   Класс Hero может также включать такие поля данных, как QString name (имя игрока), int health (уровень «жизни»), int money (деньги игрока). Этот список может быть расширен в зависимости от ваших задумок.

   Конструктор класса Hero будет получать на вход указатель на лабиринт и имя игрока. Сохраните эти параметры в соответствующих полях класса и установите начальные значения для остальных параметров.

5. Теперь наделим игрока способностью перемещаться в лабиринте.

   Создайте в классе Hero метод move(Direction direction), параметр которого задаёт направление перемещения. В методе измените номер текущей комнаты.

   Чтобы интерфейс игры мог отслеживать перемещение игрока, предусмотрим сигнал, который будет посылаться игроком при входе в комнату. Объявите в классе Hero сигнал hero_moved(int room). Аргумент сигнала будет сообщать номер комнаты, в которую входит игрок. Пошлите этот сигнал в конце метода move().

6. Создайте класс Game, который будет хранить информацию об игровой ситуации. Пока в этом классе будут два члена: hero – объект для героя и maze – объект лабиринта.
   Сделайте переменные класса Game открытыми (public), чтобы не затруднять доступ к ним из пользовательского интерфейса.

   Конструктор класса Game будет получать на вход имя игрока. Он должен инициализировать игрока, используя это имя и указатель на лабиринт.

4. Реализуйте взаимодействие модели с интерфейсом

1. Добавьте в класс формы главного окна поле данных типа Game *, которое будет хранить указатель на игровую ситуацию. Начальное значение этого поля будет передаваться главному окну через параметр конструктора.

2. Добавьте в класс главного окна слот enterRoom(int room), который будет обрабатывать вход игрока в комнату. Параметром слота является номер комнаты, в которую перемещается игрок. Слот должен:
   - вывести название и описание комнаты в соответствующие виджеты;
   - спрятать/сделать неактивными кнопки перехода для направлений, в которых нет двери (индекс соседней комнаты равен -1);
   - показать/сделать активными кнопки перехода для направлений, в которых комнаты есть.

3. Вызовите enterRoom(0) в конце конструкторе класса главного окна приложения, чтобы отобразить начальное состояние игры.

4. В функции main() с помощью QInputDialog::getText() запросите имя для игрока. Создайте объект Game и передайте его адрес в конструктор главного окна приложения.

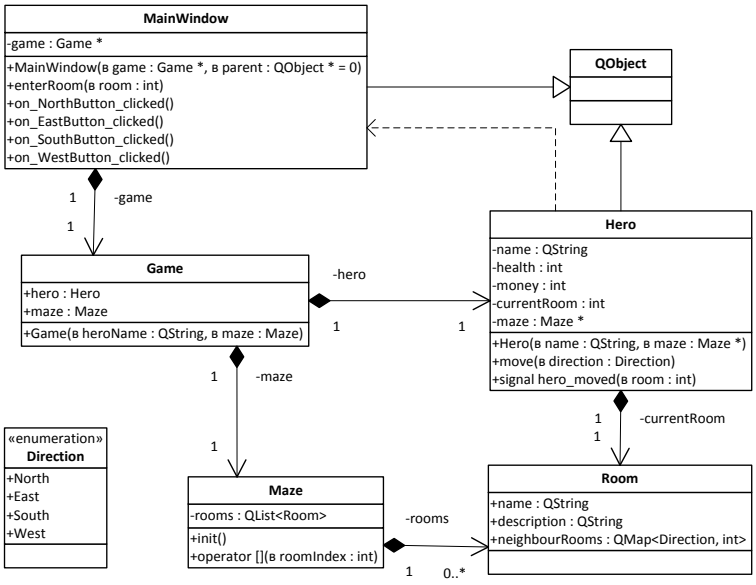
5. Запустите программу, и убедитесь, что она корректно отображает описание начальной комнаты.

6. В конце конструктора формы главного окна приложения соедините сигнал Hero::hero_moved(int) со слотом enterRoom(int).

7. Добавьте слоты для обработки нажатия на кнопки перемещения игрока. Слоты должны вызывать метод move() игрока с соответствующим параметром direction.

8. Проверьте работу игры.

Следующая схема показывает разработанные на данный момент классы приложения и связи между ними.



5. Реализуйте работу с артефактами и внутриигровую торговлю

1. Определите базовый класс Item для предметов, которые может носить игрок. Полями данных этого класса будут строки name и description, предназначенные для хранения названия и описания предмета. Их значения будут передаваться как аргументы конструктора.

   Создайте методы getName() и getDescription(), возвращающие название и описание предмета соответственно.

   Создайте пустой виртуальный метод void consume(Hero * hero), который будет реализовывать действие по использованию предмета. В классе Item он ничего не делает – он будет переопределяться в дочерних классах.

   Создайте виртуальный метод bool useOnce(), который будет сообщать, уничтожается ли данный предмет в процессе использования. Например, еду съедают и её нельзя использовать дважды. Реализация в классе Item будет возвращать false, показывая, что по умолчанию все предметы является «многоразовыми». «Одноразовые» предметы будут переопределять этот метод.

2. В классе героя Hero создайте список указателей на объекты класса Item, в котором будут храниться переносимые игроком предметы, например QList<shared_ptr<Item>> inventory.

   Добавьте герою метод addItem(shared_ptr<item>), который заносит новый предмет в список.

   Добавьте герою метод getItems(), возвращающий список имеющихся у игрока предметов.

   Определите сигнал inventory_changed(QList<shared_ptr<Item>> items), который будет уведомлять интерфейс об изменении списка предметов. Вызовите этот сигнал в конце метода addItem.

3. Определите класс Food, унаследованный от Item, представляющий предметы еды. В этом классе будет храниться значение здоровья, на которое увеличивается «жизнь» игрока при употреблении еды.

   Параметрами конструктора будут являться название, описание и прибавка к здоровью игрока. Первые два параметра передайте конструктору базового класса, последний – сохраните в поле данных.

   Переопределите метод useOnce(), чтобы он возвращал значение true.

   В конструкторе класса Game добавьте герою начальный запас еды, например, так:

   ```
   hero->addItem(make_shared<Food>(“Бутерброд”, “Увеличивает жизнь на 5”, 5));
   ```

4. Определите в классе формы главного окна слот show_inventory(QList<shared_ptr<Item>> items), который должен отобразить предметы из списка в виджете QListWidget[1]. Чтобы добавить отображение описаний предметов в виде всплывающих подсказок, можно использовать метод setToolTip() класса QListWidgetItem.

   В конце конструктора формы главного окна вызовите этот слот, передав ему полученный с помощью метода getItems() начальный список предметов игрока.

5. Проверьте, что программа правильно отображает начальный список предметов игрока.

6. Подготовьте героя к возможности траты и пополнения денег следующим образом.

   Объявите сигнал money_changed(int money), который будет уведомлять интерфейс об изменении денег.

   Добавьте метод bool changeMoney(int delta), который изменит сумму денег игрока на заданную величину. Если параметр delta отрицателен, то деньги должны уменьшится. В этом случае нужно проверить, что после вычитания у игрока не получится отрицательная сумма, и если это так – вернуть false, не изменяя сумму. Во всех остальных случаях сумма изменяется и возвращается значение true.

   Добавьте метод int getMoney(), возвращающий имеющиеся у игрока деньги.

7. Если для отображения доступных денег в главной форме игры использовать QSpinBox, то можно просто соединить сигнал money_changed от Hero со слотом QSpinBox::setValue() для отображения суммы доступных игроку денег. Установите у QSpinBox свойство readOnly, чтобы его значение нельзя было изменить руками.

   Аналогично, можно использовать виджет QLCDNumber со слотом display().

   При использовании других виджетов, например QLabel, потребуется создать слот для обработки сигналов от героя в классе формы главного окна и реализовать вывод суммы с помощью этого слота.

   В конструкторе класса формы главного окна игры выведите начальную сумму денег у

---

[1] За использование QListView будет поставлено больше баллов.
Для этого потребуется:

- унаследовать класс Hero от класса QAbstractListModel (или QAbstractListModel, если будет использоваться таблица);
- определить в классе Hero метод rowCount(), который будет возвращать число предметов игрока;
- определить в классе Hero метод data(), который будет возвращать название предмета;
- для таблиц определить методы columnCount() и headerData());
- в методах, изменяющих набор инвентаря героя (например, addItem()) перед модификацией списка вызывать beginResetModel(), а после неё – endResetModel();
- в конструкторе класса главной формы игры вызвать setModel(hero) чтобы соединить представление и модель.

При использовании QListView не потребуется делать сигнал inventory_changed() у героя и слот show_inventory() в классе формы главного окна приложения. Вместо них будут использоваться сигналы модели, активируемые функциями beginResetModel() и endResetModel().
Вместо beginResetModel()/endResetModel() лучше использовать более конкретные функции уведомления, такие как beginInsertRows()/endInsertRows(), beginRemoveRows()/endRemoveRows() или сигнал dataChanged(). Вариант …Reset… является наиболее общим уведомлением.

игрока в соответствующий виджет.

8. Создайте новый класс формы ShopWindow для реализации магазина, унаследовав её от диалогового окна. В конструктор класса добавьте параметр для передачи указателя на героя.

   Предусмотрите в форме поля’ для вывода имеющихся у героя денег, списка возможных покупок, кнопки «купить» и «закрыть».

   Список возможных покупок может быть реализован с помощью QTableWidget или QTableView[2]. Заполните таблицу данными о предметах, которые будут продаваться в магазине. Для QTableWidget это можно сделать в конструкторе диалога.

   Для вывода доступных денег используйте такой же способ, как в главном окне игры.
   Соедините кнопку «закрыть» со слотом accept(), чтобы обеспечить закрытие окна.

9. Добавьте в главное окно кнопку для вызова магазина (openShop).

   Определите, в какой момент игроку будет доступен магазин: в любое время или когда он находится в определённой комнате. В последнем случае можно ввести в класс Room соответствующий флаг, показывающий доступность магазина, и активировать кнопку в слоте enterRoom(). Можно сделать магазин доступным для начальной (нулевой) комнаты.

   Добавьте для этой кнопки слот (on_openShop_clicked), в котором создайте и запустите диалог магазина.

10. Проверьте, что запуск магазина и отображение доступных денег работают правильно.

11. В классе формы магазина создайте слот для обработки нажатия на кнопку покупки (on_buy_clicked).

    Слот должен списать у героя денег, вызвав метод changeMoney(), и, если этот метод вернул true, создать и добавить игроку соответствующий предмет.

    Чтобы определить, какой предмет продавать, определите строку с текущей ячейкой:

```
QModelIndex currentIndex = ui->shopTable->currentIndex();
if(!currentIndex.isValid())
    { return; // нет текущей ячейки, ничего не делаем }
int currentRow = currentIndex.row();  // номер строки с текущей ячейкой
```

12. Проверьте корректность работы магазина.

13. Теперь обеспечьте возможность использования первого типа предметов – еды.

    Определите в герое сигнал health_changed(int health), уведомляющий об изменении уровня здоровья.

    Определите методы int getHealth() и void changeHealth(int health_delta), которые

[2] За использование QTableView будет поставлено больше баллов.

возвращают и изменяют уровень здоровья. Проверку на снижение уровня здоровья до нуля пока можно не делать. Реализуйте ограничение на максимальный уровень здоровья.

14. Реализуйте в главном окне игры вывод уровня здоровья, это можно сделать аналогично выводу денег, но в данном случае удобно использовать виджет QProgressBar.

15. Добавьте в класс Food реализацию метода void consume(Hero *hero), которая вызовет метод changeHealth() героя и добавит ему здоровья.

16. Добавьте в класс Hero метод useItem(int index), который должен реализовать употребление предмета, заданного индексом в списке.

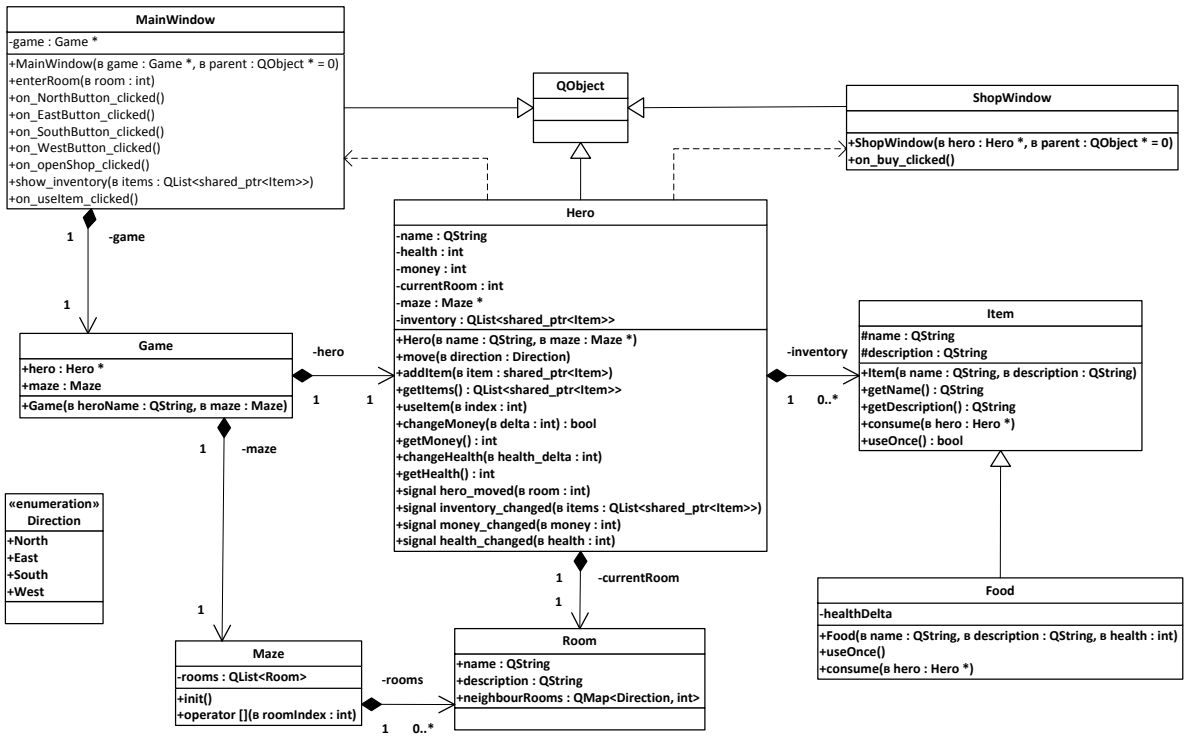
    Вызовите метод consume() для указанного предмета, передав ему указатель на героя (this) в качестве аргумента. После этого вызовите метод useOnce(), чтобы проверить, не являлся ли предмет одноразовым. Если предмет одноразовый – удалите соответствующий элемент из списка и активируйте сигнал inventory_changed().

17. Добавьте в форму главного окна кнопку «использовать предмет» (useItem). Создайте слот для обработки нажатия на эту кнопку (on_useItem_clicked).

    Используйте currentIndex(), чтобы получить номер текущей записи в списке предметов (аналогично реализации покупки в магазине). Получив индекс, вызовите метод useItem() героя.

18. Запустите программу и проверьте, что употребление еды работает корректно. Здоровье игрока должно увеличиваться, а еда – исчезать из списка предметов.

На рисунке показаны классы игры после внесённых изменений.



6. Добавьте возможность собирать предметы в лабиринте

1. Добавьте в класс Room список указателей на предметы, которые лежат в этой комнате, например QList<shared_ptr<items>> items.

   Добавьте в класс Room методы:

   void putItem(shared_ptr<Item> item) – добавляет предметы в список;

   QList<shared_ptr<Items>> visibleItems(Hero *hero) – возвращает список предметов, видимых для героя (пока будет возвращать весь список, в будущем можно реализовать возможность не показывать все или часть предметов герою в зависимости от каких-то условий, например наличия у игрока включённого фонаря или зажженного факела);

   shared_ptr<Item> peekItem(int index) – вынимает заданный индексом предмет из списка и возвращает его (предмет должен быть убран из списка).
2. В методе Init() класса Maze добавьте несколько предметов еды в лабиринт.

3. В форме главного окна приложения вставьте виджет QListWidget для вывода списка видимых в комнате предметов и кнопку «подобрать предмет» (peekItem).

   В слоте enterRoom() выведите названия предметов, полученных через visibleItems() в список. Делайте кнопку peekItem активной или не активной в зависимости от того, пуст этот список или нет.

4. Сделайте класс Item наследником класса std::enable_shared_from_this<Item> следующим образом:

   ```
   class Item : public std::enable_shared_from_this<Item>
   {
       …
   ```

   Добавьте в класс Item метод void peek(Hero *hero), который будет вызываться в момент, когда герой подберёт этот предмет. В этом методе добавьте текущий предмет в инвентарь героя следующим образом[3]:

   ```
   hero->addItem(shared_from_this());
   ```

5. В классе Game добавьте метод void peekItem(Hero *hero, int itemIndex), задачей которого является реализация логики взятия предмета из текущей комнаты.
   Необходимо вынуть предмет из комнаты и вызвать для него метод peek.

6. Добавьте в класс формы главного окна слот для обработки нажатия на кнопку «подобрать предмет» (on_peekItem_clicked). Слот должен определить индекс выделенной записи в списке видимых предметов и вызвать метод peekItem() у game.

7. Проверьте работу программы. Игрок должен получить возможность собирать предметы.

[3] Метод shared_from_this() позволяет создать «правильный» shared_ptr, указывающий на «себя». Для этого нельзя использовать просто shared_ptr<Item>(this), так как это приведёт к неправильному подсчёту ссылок и уничтожению объекта раньше, чем нужно.

8. Теперь добавьте новый класс предметов, которые игрок может находить в игре – золото.

   Создайте класс Gold, унаследованный от класса Item. В поле данных amount этого класса будет хранится количество денег.

   Конструктор класса будет получать аргументы name – название предмета, description – описание и amount – количество денег.

   Переопределите в классе Gold метод peek(Hero * hero). Этот метод не должен добавлять ничего в список предметов игрока, вместо этого он должен вызвать changeMoney() чтобы добавить игроку денег.

9. В методе Init класса Maze разместите в лабиринте несколько экземпляров денег.

10. Проверьте работу программы. При сборе денег они не должны появляться в списке предметов героя. Вместо этого должно увеличиваться количество денег.

7. Реализуйте врагов и сражения

1. Создайте два новых класса предметов: оружие Weapon и щит Shield.

   Weapon будет хранить уровень урона int damageLevel и содержит метод int getDamage(), который его возвращает. Уровень урона устанавливается вместе с названием и описанием через параметры конструктора.

   Shield хранит уровень защиты int shiledLevel, доступ к которому предоставляется методом int getShiled(). Конструктор получает на вход название, описание и уровень защиты.

   Разместите новые предметы в лабиринте или сделайте их доступными для покупки через магазин.

2. Запустите игру и проверьте, что игрок может получать новые предметы.

3. Теперь подготовьтесь к созданию класса врагов. Враги будут иметь некоторые общие свойства с героем –у них будет имя, уровень здоровья и методы для манипуляций с ним. Поэтому можно создать для героя и врагов общий суперкласс.

   Создайте класс Actor, унаследованный от QObject. Перенесите в этот класс из класса Hero переменные name и health, методы changeHealth(), getHealth, сигнал health_changed().

   Конструктор класса Actor будет получать в качестве параметра имя и указатель на родителя QObject *parent[4].

   Измените у класса Hero родительский класс с QObject на Actor.

4. Скомпилируйте программу и проверьте, что её работоспособность не нарушена.

[4] Это позволит организовать автоматическое удаление врагов.

5. Добавьте в класс Actor чистый виртуальный метод int getShiled().

   Сделайте реализацию этого метода в классе Hero. Он должен просмотреть список предметов игрока, с помощью dynamic_cast<Shield> найти в нём все щиты, определить и вернуть максимальный уровень защиты среди них. Если ни одного щита не найдено, следует вернуть некоторое минимальное значение, которое будет соответствовать уровню защиты игрока без щитов.

6. Добавьте в класс Actor чистый виртуальный метод int getDamage(). Перегрузите этот метод в классе Hero аналогично методу getShield().

7. Создайте класс Enemy, унаследованный от Actor. Не забудьте, что как все унаследованные от QObject классы он должен содержать строку Q_OBJECT в начале своего определения.

   Поля данных этого класса включают: shiledLevel – уровень защиты, damageLevel – уровень нападения и вероятность нападения strikeProbability, которая будет определять, как часто враг будет нападать на игрока.

   Реализуйте методы int getDamage(), int getShield() и int getStrikeProbability() возвращающие уровень нападения, уровень защиты и вероятность нападения соответственно.

   Конструктор класса Enemy должен получать на вход имя; уровни защиты и нападения; вероятность нападения; родителя – QObject *parent.

8. Сделайте класс Game потомком QObject.

   Создайте в этом классе отображение, которое будет хранить информацию о том, в каких комнатах сидят враги: QMap<Room*, Enemy* > Enemies[5].

   Добавьте метод void addEnemy(Room *room, Enemy *enemy) для внесения врагов в это отображение.

9. Добавьте методу Init класса Maze параметр Game *game.

   В этом методе создайте несколько врагов и добавьте их в игру. При создании врагов обязательно используйте оператор new и указывайте game как родителя. Это обеспечит автоматическое уничтожение объектов Enemy при уничтожении объекта game.

10. Теперь обеспечьте передачу информации о перемещении игрока между комнатами в класс Game.

    Создайте в классе Game слот heroMoved(int room). В конструкторе класса Game соедините этот слот с сигналом hero_moved(int room) героя.

    В слоте проверьте, нет ли врага в комнате, в которую попал игрок. Если есть – надо начинать битву.

[5] Используйте QMultiMap, если хотите, чтобы в одной комнате могли находиться сразу несколько врагов.

Создайте в классе game метод void battle(Hero *hero, Enemy *enemy). Вызовите этот метод из слота heroMoved(), в момент когда выявлена встреча игрока с врагом.

11. Создайте класс Battle, унаследованный от QObject.

    В полях класса будут хранится указатели на героя Hero *hero и врага Enemy *enemy. Их значения задаются через конструктор класса. Создайте методы getHero() и getEnemy() для доступа к полям.

12. Объявите в классе Game сигнал void battle_started(Battle *battle), он будет извещать интерфейс о том, что сражение началось.

    В методе battle() создайте объект Battle и активируйте сигнал battle_started.

13. Создайте класс формы для боя BattleDialog, унаследованный от QDialog.

    К параметрам конструктора добавьте Battle *battle, и сохраните его в поле класса.

    Разработайте интерфейс формы. Он должен отображать имена сражающихся, их уровни жизни. Для героя сделайте кнопки нанесения удара и побега с места боя.

    В конструкторе выведите имена сражающихся в соответствующие виджеты. Свяжите сигналы изменения уровня жизни с виджетами для их отображения.

14. В классе формы главного окна приложения создайте слот void battleStarted(Battle *battle).

    В этом слоте создайте диалог BattleDialog и запустите его в модальном режиме, используя exec().

15. Запустите программу. Убедитесь, что окно боя отображается в нужный момент и отображает информацию.

16. В классе Battle объявите сигнал battle_hit(Actor *from, Actor *to, int damage), который сообщает, что персонаж from нанёс персонажу to удар, унесший здоровья в размере damage.

    Объявите слот hero_attack(), реализацию пока оставьте пустой. Этот слот будет вызываться из интерфейса, когда игрок будет наносить удар.

17. В форме BattleDialog создайте виджет, с помощью которого будут отображаться сообщения об ударах. Это может быть QLabel, который будет отображать последнее сообщение, или QListWidget, в котором будет показана вся история боя.

    Создайте в классе BattleDialog слот battleHit(Actor *from, Actor *to, int damage). В слоте выведите информацию об ударе в созданный виджет.

    В конструкторе BattleDialog соедините сигнал battle_hit() от battle со слотом battleHit() и сигнал от кнопки «ударить» со слотом hero_attack() в battle.

18. Добавьте в класс Battle метод hitDamage(int hitLevel, int shieldLevel), который будет вычислять урон, нанесённый здоровью персонажа с уровнем щита shiledLevel, если его ударили с силой hitLevel.

    Можно провести вычисление, например, следующим образом. Вычислите максимальную силу удара, вычтя shieldLevel из hitLevel. Проверьте, что результат не меньше порогового значения, если меньше – установите максимальную силу удара равной порогу. Сгенерируйте случайное число из диапазона от минимального значения до рассчитанного максимального. Верните это случайное число. Параметры минимальной силы удара и порога задайте по своему желанию.

19. Добавьте в класс Battle метод attack(Actor *from, Actor *to), который обрабатывает удар, нанесённый персонажем from персонажу to.

    В этом методе получите значение getDamage() от персонажа from и getShield() от персонажа to. На их основе вычислите с помощью метода hitDamage() нанесённый урон. Активируйте сигнал battle_hit(). Уменьшите значение здоровья персонажа to на значение урона.

    Скорректируйте метод changeHealth() в классе Actor чтобы он не давал уменьшить уровень здоровья ниже 0.

    В слоте hero_attack() класса Battle вызовите attack(hero, enemy).

20. Запустите программу и начните бой. При нажатии на кнопку удара должны отображаться уведомления и уменьшаться уровень здоровья врага.

21. Пока враг не причинял вреда герою. Исправим это.

    Добавьте поле int timerID в класс Battle. В конструкторе создайте таймер с интервалом 500 мс и запомните его идентификатор в timerID.

    Добавьте метод void timerEvent(QTimerEvent *event). Сгенерируйте случайное число в некотором диапазоне и сравните его с enemy->strikeProbability(). Если сгенерировано меньшее число – вызовите attack(enemy, hero).

    Если генерируются числа в диапазоне 0..100, а strikeProbability равно 50, то враг будет нападать на героя в среднем каждый второй такт таймера. Если strikeProbability равно 25 – то каждое четвёртое.

22. Запустите программу. Убедитесь, что враг стал наносить удары герою.

23. Теперь нужно зафиксировать факт окончания боя.

    Добавьте в класс Actor метод bool isAlive(), возвращающий true если уровень жизни персонажа больше 0.

    Объявите в классе Battle сигнал battle_finished(Actor *winner, Actor *looser). В методе attack() после уменьшения уровня здоровья атакованного проверьте, жив ли он. Если нет –

остановите таймер (использовав сохранённое значением timerID) и активируйте сигнал battleFinished(from, to).

24. В классе формы BattleWindow создайте слот void battleFinished(Action * winner, Action * looser). Соедините этот слот с сигналом battle_finished от battle.

    В слоте выведите информацию о завершении боя. Можно использовать QMessageBox или вывести текст в каком-нибудь виджете самого окна. В первом случае можно закрыть окно BattleWindow сразу после уведомления. Во втором – предусмотрите кнопку, нажатие которой закроет окно. Эта кнопка должна быть неактивна или спрятана до окончания боя.

25. В классе Game объявите сигнал void game_over(bool result).

    В конце метода battle() класса Game, проверьте, выжил ли враг после сражения. Если нет – удалите его из отображения enemies. Можно создать специальный класс предметов для тел врагов (если для героя есть смысл собирать их), или поместить на место врага какие-то полезные предметы – его оружие или сокровища[6].

    Теперь проверьте, выжил ли сам герой. Если нет – активируйте сигнал game_over() с параметром false.

26. Реализуйте в классе формы главного окна игры слот void gameOver(bool result).

    В слоте покажите сообщение о результатах игры. Если главное окно останется открытым, отключите все органы управления игрой.

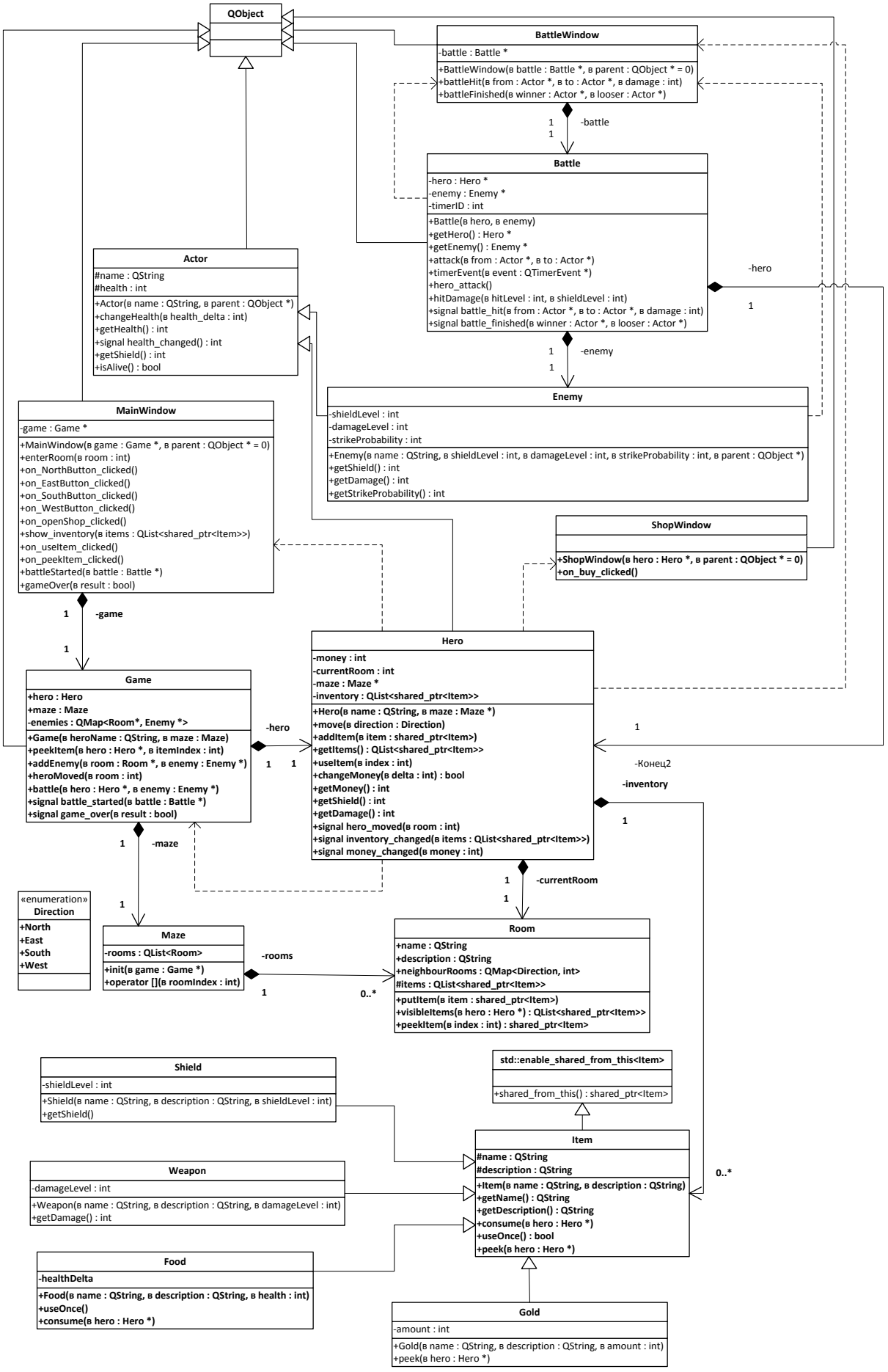
27. Запустите игру и проверьте её работу.

28. Теперь сделайте возможным выигрыш. Можно добавить в слот heroMoved() класса Game, проверку, что игрок достиг некоторой комнаты, и, возможно, имеет с собой определённый предмет или предметы. Комнату можно пометить флагом, хранящимся как переменная класса Room. Если условие выигрыша выполнено – активируйте сигнал game_over() с параметром true.

29. Проверьте работу законченной версии игры. Подберите значения параметров так, чтобы в неё было интересно играть.

[6] Так как сигнал hero_moved соединяется со слотом heroMoved() в классе Game раньше, чем со слотом enterRoom() в классе MainWindow, то бой запустится раньше, чем в главном окне будет показано содержимое новой комнаты. Слоты для одного сигнала вызываются в том порядке, в котором они были подключены. Поэтому модификация списка предметов, находящихся в комнате, после боя не потребует отдельных уведомлений в интерфейс.

На рисунке показана окончательная диаграмма классов игры.



Возможные дополнения к игре

Можно внести в игру различные дополнения, которые могут сделать её более интересной. Ниже представлены некоторые идеи, которые можно реализовать.

Добавьте возможность для игрока покинуть поле боя до окончания сражения. Если герой выбирает «бежать» его можно вернуть в предыдущую комнату или в случайную соседнюю комнату. Можно сделать у врагов флаг, показывающий можно убегать от такого врага или нет.

В окне боя можно отобразить инвентарь героя. Это может позволить: выбирать оружие для удара или щит для защиты. Поправлять здоровье во время битвы. Использовать волшебные амулеты дающие возможность покинуть поле боя досрочно, даже если враг не позволяет.

Можно сделать, чтобы разные виды оружия действовали на разных врагов по-разному или не действовали вообще.

Можно уменьшать жизнь героя на небольшую величину при каждом перемещении между комнатами. Это не даст ему гулять бесконечно без запаса еды.

Можно сделать «отравленные» комнаты, пребывание в которых уменьшает жизнь героя по таймеру, или сразу убивающие его при входе.

Можно сделать двери, проход через которые требует наличия у игрока ключа, заклинания или волшебного амулета. При попытке пройти через такую дверь без ключа можно показать сообщение, поясняющее, почему проход не возможен.

Можно реализовать возможность выкладывать предметы в комнатах. Ограничить число, размер или вес переносимых предметов. Сделать предмет «тележку», помогающий преодолеть это ограничение.

Можно реализовать различные взаимодействия между предметами. Если у игрока есть факел и огниво он может зажечь факел. Наличие зажженного факела может быть обязательным, чтобы увидеть содержимое некоторых комнат.

Можно сделать возможным взаимодействие предметов в комнате. Если в комнате со старым глухо запертым сундуком положить горящий факел, то сундук может сгореть, а на его месте остаться меч.

Можно дать возможность врагам перемещаться по лабиринту, случайно или по установленному маршруту.

Можно населить лабиринт нейтральными существами, которые не наносят вреда герою. Некоторых из них можно приручить и отдавать команды следовать за героем, остаться на месте или идти гулять. Эти существа могут помогать герою в бою с врагами или выполнять другие функции.

Можно добавить в игру роботов, которые могут выполнять заданные программы: двигаться и переносить предметы. Если в игре реализована возможность делать «ключи» для дверей и другие ограничения, то наличие роботов позволит создавать сложные головоломки. Роботов также можно приспособить для прохода через «команты смерти» и исследование местности за ними.

Можно реализовать возможность продажи предметов в магазине.

Игровых магазинов может быть несколько, в них могут продавать разные предметы и по разным ценам.