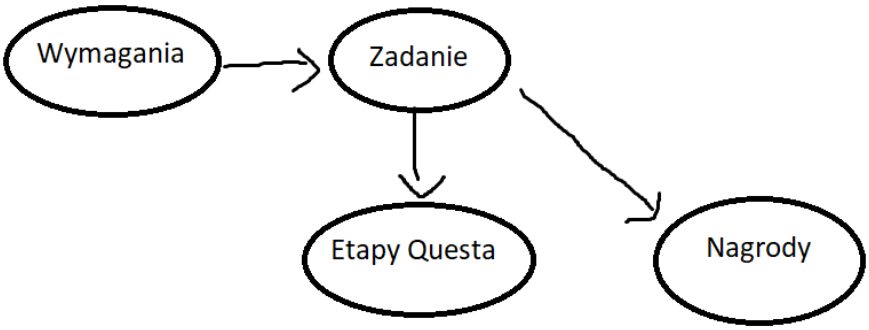
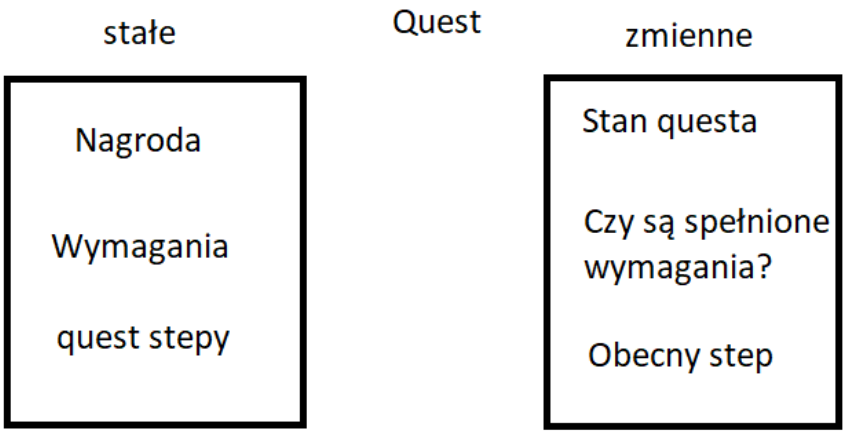
Questy składają się z takich elementów jak:

- Wymagania, które muszą być spełnione, aby ten quest mógł zostać rozpoczęty. W tym wypadku wymaganiem może być ukończenie innego zadania, albo ilość zgromadzonych punktów
- Samo odniesienie do questa (reprezentowany np. Przez jego id)
- Poszczególne etapy questów, nazywane “quest step”
- Nagroda za zadanie – w tym systemie nagrodą za zadania są punkty, które mogą być także wspomnianym wcześniej wymaganiem

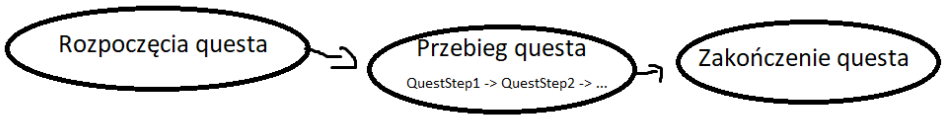


Z kolei informacje, które zawiera zadanie dzielą się na dwa rodzaje: stałe oraz zmienne.



Z tego powodu najlepszą decyzją byłoby rozdzielenie tych dwóch rodzajów informacji - stałe informacje, niezmienne w czasie mogą być zawarte w scriptable objectach. Pozwoli to na łatwy dostęp do informacji, bezproblemowe stworzenie nowych questów. Z kolei informacje, które będą się zmieniać wraz z questem zostaną umieszczone wewnątrz klasy.

Questy składają się z 3 części:



Rozpoczęcia questa jest tak naprawdę najprostszą częścią. Rozpoczęcia questa to w rzeczywistości wymaga jedynie sprawdzenia warunków do rozpoczęcia. Jeśli w trakcie interakcji ze zleceniodawcą, zostaną spełnione warunki, gra powinna od razu przejść do pierwszego quest stepu.

Quest step może być dowolną rzeczą, dlatego powinien zostać stworzony najogólniej jak się da. Z tego powodu system nie będzie “pytał” o żadne warunki – te warunki powinny być ustalane wewnątrz samego stepu. Z tego powodu każdy quest step będzie osobnym prefabem tworzonym na scenie (wyciąganym z puli). To pozwoli na łatwy dostęp do informacji wraz z przechowywaniem zmieniających się w czasie danych jak przykładowo licznik zdobytych monet, pozwalając na zakończenie stepu, gdy te monety osiągną odpowiedni poziom.

Ostatnim etapem jest zakończenie questa. Quest zostaje zakończony, gdy wszystkie quest stepy zostaną zakończone. Trzeba mieć na uwadze, że quest może, ale nie musi wymagać powrotu do zleceniodawcy. Trzeba wziąć pod uwagę, że questy bywają zakończone “w powietrzu” - np. Podniesienie przedmiotu natychmiast kończy zadanie.

Scriptable obiekt zawierający statyczne informacje:

```
public class QuestInfo : ScriptableObject
{
    [field: SerializeField]
    Odwołania: 11
    public string id { get; private set; }
    public List<QuestInfo> questRequirements;
    public int scoreRequirement;
    public List<GameObject> questSteps;
    public int pointsReward;
}
```

Fragment klasy Quest zawierająca zmienne informacje:

```
Odwołania: 19
public class Quest
{
    public QuestInfo questInfo;
    public QuestState questState;
    private int currentQuestStepIndex;

    public ObjectPool<GameObject> questStepPool;

    1 odwołanie
    public Quest(QuestInfo questInfo)
    {
        this.questInfo = questInfo;
        this.currentQuestStepIndex = 0;
        this.questState = QuestState.CantStart;
    }
```

Każde zadanie składa się z poszczególnych etapów. Etapy te są stworzone następująco:

QuestStep jest klasą abstrakcyjną, z której powinny korzystać wszystkie quest stepy. Taka konstrukcja pozwala zachować porządek z kodzie. Każdy rodzaj quest stepu ma takie elementy jak zakończenie stepu, toteż nie ma sensu przepisywać tego samego do innych obiektów. Dodatkowo w miarę rozrastania się zadań, może się okazać, że któryś quest step będzie działał nieco inaczej, niż zakłada wersja bazowa. Wtedy klasa abstrakcyjna pozwoli na obsługę wyjątkowej sytuacji bez komplikowania kodu.

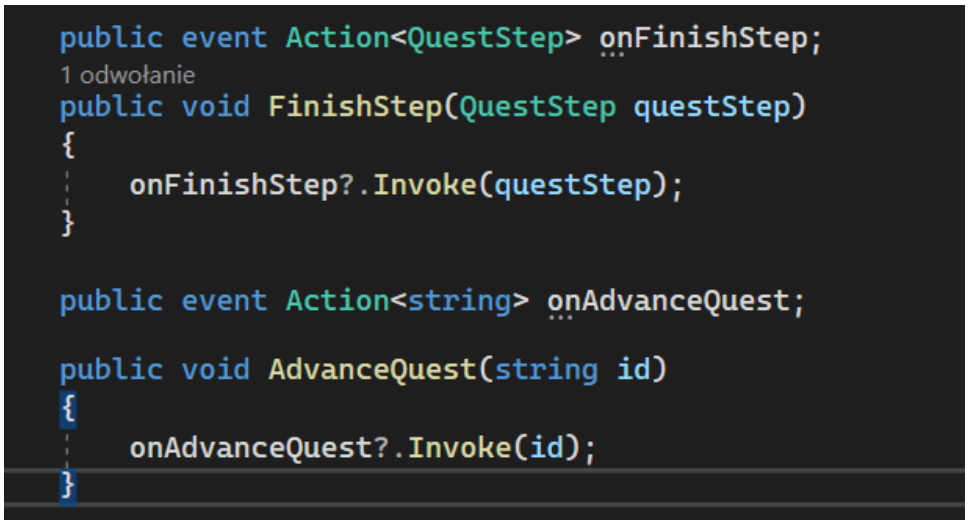
Klasa zawiera dwie metody wspólne dla każdego stepu:

```
1 odwołanie
protected void FinishQuestStep()
{
    if (!isFinished)
    {
        isFinished = true;
        GameEventsManager.instance.questEvents.AdvanceQuest(questId);
        GameEventsManager.instance.questEvents.FinishStep(this);
    }
}

1 odwołanie
public void InitializeQuestStep(string questId)
{
    this.questId = questId;
}
```

InitializeQuestStep jest pośrednikiem, który informuje QuestStep o tym, którego questa jest elementem. Choć można zapisać id w publicznym polu w edytorze, taki zapis pozwoliłby w przyszłości przypisać większą ilość danych potrzebnych przy inicjalizacji.

FinishQuestStep zawiera dwa eventy: jeden, który porusza quest do przodu, oraz drugi, który jest konterem na metody, które powinny dziać się po zakończeniu stepu (przykładowo powrót prefabu z QuestStep do puli)

Oba eventy korzystają z customowego managera eventów



Umieszczenie każdego eventu w managerze zmniejszy znacząco ilość zależności. W dużym projekcie zmniejszy to ilość sphagetti codu, a nawet w takim małym znacznie ułatwi dostęp do eventów - zamiast korzystać z wielowarstwowych referencji mamy dostęp do eventu dzięki singletonowi.

Każdy quest zawiera 3 główne eventy:

```
public event Action<string> onStartQuest;
1 odwołanie
public void StartQuest(string id)
{
    onStartQuest?.Invoke(id);
}

public event Action<string> onAdvanceQuest;
1 odwołanie
public void AdvanceQuest(string id)
{
    onAdvanceQuest?.Invoke(id);
}

public event Action<string> onFinishQuest;
1 odwołanie
public void FinishQuest(string id)
{
    onFinishQuest?.Invoke(id);
}
```

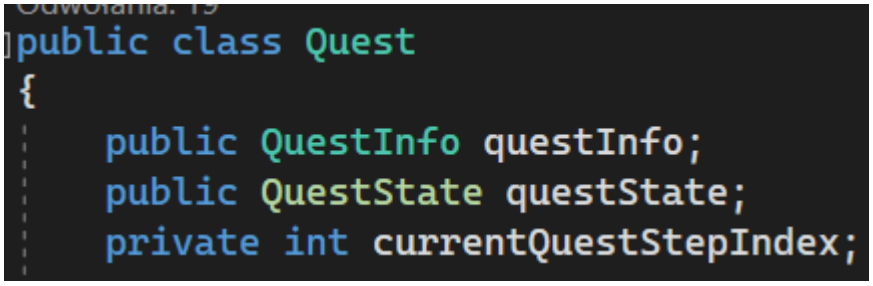
Całością systemu zarządza Quest Manager. To w nim używane są metody startu, progresowania i zakończenia questów. To tu także zawarta jest lista wszystkich obecnych questów. Jest to system obsługujący poszczególne etapy zadania. Zawiera on ogólną interpretacje poszczególnych etapów - przykładowo zakończenie zadania daje nagrodę, a rozpoczęcie tworzy nowy quest step. Dopiero kolejne skrypty decydują o “szczegółach”. Takie podejście rozdziela odpowiedzialność skryptów unikając sytuacji, gdzie jeden element kodu decyduje o wielkiej ilości poszczególnych funkcjonalności. Zostawienie jednak ogólnego przebiegu zadań w jednym managerze zmniejsza skomplikowanie całego systemu nie zaburzając rozdzielenia odpowiedzialności.

W tym miejscu zapełniany jest słownik zawierający zadania.

```
1 odwołanie
private Dictionary<string, Quest> CreateQuestMap()
{
    QuestInfo[] allQuests = Resources.LoadAll<QuestInfo>("Quests");
    Dictionary<string, Quest> idToQuestMap = new Dictionary<string, Quest>();
    foreach (QuestInfo questInfo in allQuests)
    {
        if (idToQuestMap.ContainsKey(questInfo.id))
        {
            Debug.LogWarning("Duplicate ID found when creating quest map: " + questInfo.id);
        }
        idToQuestMap.Add(questInfo.id, new Quest(questInfo));
    }
    return idToQuestMap;
}
```

Questy zawarte są w słowniku. Struktura ta pozwala na stworzenie listy questów z przyporządkowanymi im informacjami. Ułatwia to prosty odczyt każdej informacji dotyczącej questa znając tylko id danego zadania.

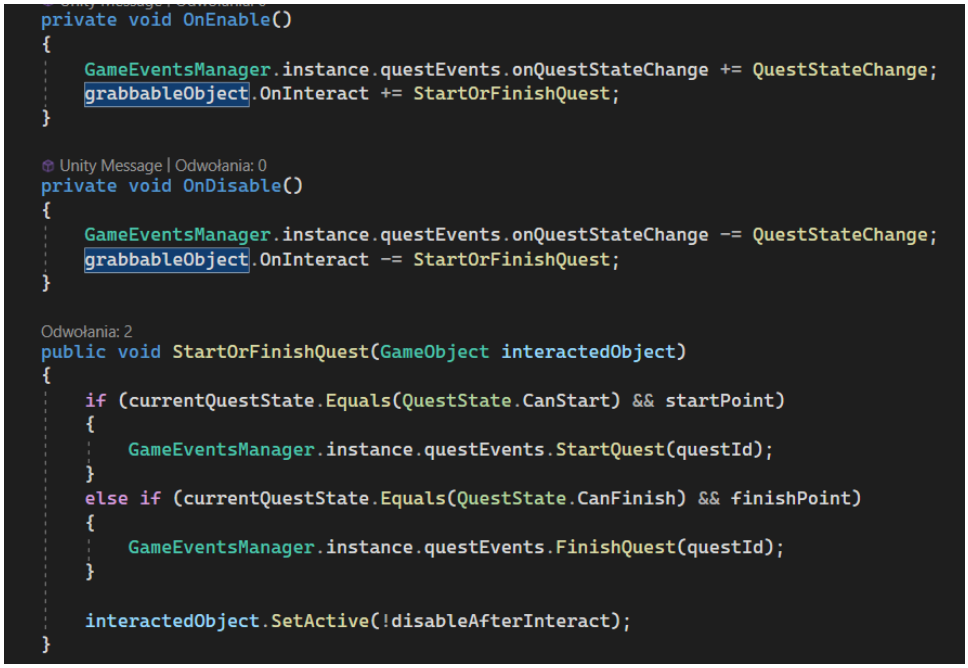
Klasa Quest zawiera wszystkie informacje dotyczące zadań, których nie powinno się zawierać w scriptable obiektach – czyli informacji, które zmieniają się w czasie, takich obecny step Questa, albo stan questa (czy quest jest w trakcie, czy jest zakończony, albo możliwy do akceptacji)



```
public class Quest
{
    public QuestInfo questInfo;
    public QuestState questState;
    private int currentQuestStepIndex;
```

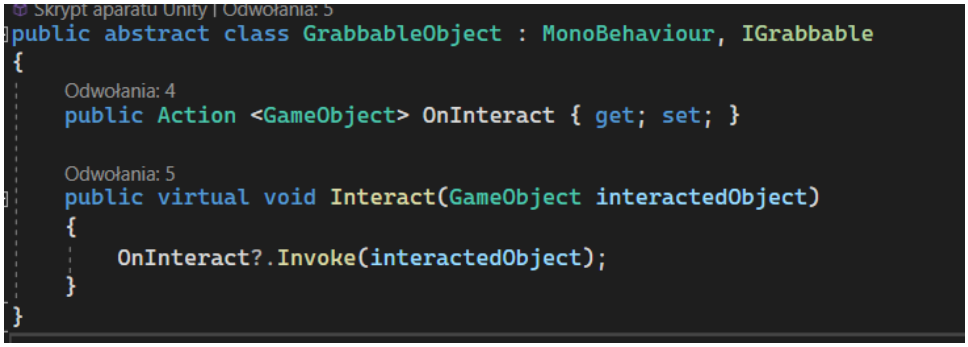
Aby jednak quest został rozpoczęty i potem obsłużony przez QuestManagera musi on zawierać w sobie skrypt QuestPoint. Ten skrypt ustala punkt początkowy, oraz końcowy każdego zadania.

Metodą odpowiadającą bezpośrednio za zakończenie lub rozpoczęcia zadania jest StartOrFinishQuest



```
private void OnEnable()
{
    GameEventsManager.instance.questEvents.onQuestStateChange += QuestStateChange;
    grabbableObject.OnInteract += StartOrFinishQuest;
}

Unity Message | Odwołania: 0
private void OnDisable()
{
    GameEventsManager.instance.questEvents.onQuestStateChange -= QuestStateChange;
    grabbableObject.OnInteract -= StartOrFinishQuest;
}

Odwołania: 2
public void StartOrFinishQuest(GameObject interactedObject)
{
    if (currentQuestState.Equals(QuestState.CanStart) && startPoint)
    {
        GameEventsManager.instance.questEvents.StartQuest(questId);
    }
    else if (currentQuestState.Equals(QuestState.CanFinish) && finishPoint)
    {
        GameEventsManager.instance.questEvents.FinishQuest(questId);
    }

    interactedObject.SetActive(!disableAfterInteract);
}
```

Jest on wywołany w momencie wejścia w interakcje z GrabbableObjectem.



```
Skrypt aparatu Unity | Odwołania: 5
public abstract class GrabbableObject : MonoBehaviour, IGrabbable
{
    Odwołania: 4
    public Action <GameObject> OnInteract { get; set; }

    Odwołania: 5
    public virtual void Interact(GameObject interactedObject)
    {
        OnInteract?.Invoke(interactedObject);
    }
}
```

Grabbable object jest klasą abstrakcyjną. Jest to klasa bazowa dla każdego obiektu w świecie gry, który można podnieść. Tyczy się to przykładowo skrzynki, albo przedmiotów kolekcjonerskich. Zawiera on w sobie interface. Z racji małego projektu taka konstrukcja mogłaby się wydawać zbędna, jednak w miarę rośnięcia projektu tworzenie obiektów przy pomocy takich “klocków” zmniejszyłaby ilość sphagetti, gdyż zamiast kolejnych stopni dziedziczenia (gdzie to dziedziczenie stale miałoby coraz głębsze poziomy) obiekty mogłyby po prostu dostawać kolejny “klocek” w postaci kolejnego interface’u.

StartOrFinishQuest, oprócz sprawdzania boola na obiekcie questowym, sprawdza także jaki jest stan obecnego questa. Stanem tym zarządza QuestManager.

```
Unity Message | Odwołania: 0
private void Update()
{
    foreach (Quest quest in questMap.Values)
    {
        if (quest.questState == QuestState.CantStart && IsRequirementMet(quest))
        {
            ChangeQuestState(quest.questInfo.id, QuestState.CanStart);
        }
    }
}
```

Ten fragment kodu zmienia stan questa na możliwy do zaczęcia sprawdzając czy są spełnione warunki. Jest to bardzo prosty sposób na zmianę stanu. W przypadku większych projektów przestanie się on sprawdzać z racji rosnącej ilości wymagań questów oraz samej liczby zadań. Gdyby projekt się rozrósł należałoby przenieść sprawdzanie wymagań zadań na sam moment, gdy jeden z elementów listy wymagań uległby zmianie. To pozwoliłyby usunąć ten fragment kodu z kłopotliwego Update’a.

Gdy gracz wejdzie w interakcje z obiektem zlecającym zadanie to przy spełnieniu warunków otrzymuje zadanie.

```
Odwołania: 2
private void StartQuest(string id)
{
    Quest quest = GetQuestById(id);
    quest.CreateQuestStep(transform);
    ChangeQuestState(quest.questInfo.id, QuestState.InProgress);
}
```

Zadanie zaczyna się do rozpoczęcia pierwszego kroku. Chodź kod nie przewiduje zaczęcia wielu stepów jednocześnie, należy pamiętać, że warunek czym jest dany step jest ustalany dopiero osobno w prefabie.

```
Odwołania: 3
public void CreateQuestStep(Transform parentTransform)
{
    GameObject questStepPrefab = GetQuestStep();

    if (questStepPrefab != null)
    {
        questStepPool = new ObjectPool<GameObject>(createFunc: () => Object.Instantiate(questStepPrefab, parentTransform),
actionOnGet: (obj) => obj.SetActive(true), actionOnRelease: (obj) => obj.SetActive(false), collectionCheck: false);

        QuestStep questStep = questStepPool.Get().GetComponent<QuestStep>();
        questStep.InitializeQuestStep(questInfo.id);
        GameEventsManager.instance.questEvents.onFinishStep += ReturnToPool;
    }
}
```

Ponieważ quest step jest prefabem, zastosowano Unity Object Pooling. W dłuższej perspektywie pozwoliłoby poprawić nieco wydajność gry, ale co ważniejsze, pozwala uniknąć stopniowego zapełniania GarbageCollectora, gdyby zdecydowane się na niszczenie obiektów, zamiast zwracania do puli.

```
Skrypt aparatu Unity (1 odwołanie do zasobu) | Odwołania: 0
public class FindChestQuestStep : QuestStep
{
    Unity Message | Odwołania: 0
    private void OnEnable()
    {
        GameEventsManager.instance.questEvents.onChestCollected += ChestCollected;
    }

    Unity Message | Odwołania: 0
    private void OnDisable()
    {
        GameEventsManager.instance.questEvents.onChestCollected -= ChestCollected;
    }

    Odwołania: 2
    private void ChestCollected()
    {
        FinishQuestStep();
    }
}
```

W tym przypadku naszym stepem jest znalezienie ukrytej skrzyni ze skarbem. Wtedy step zostaje zakończony. Oznacza to tyle, że gracz musi podnieść na mapie właściwą skrzynkę. Odpowiedzialność za zdefiniowanie co oznacza zakończenie stepu leżu po stronie prefaba. Takie rozdzielenie odpowiedzialności sprawia, że przy tworzeniu kolejnych stepów nie musimy zupełnie brać pod uwagę głównego managera. W razie, gdyby kolejne stworzone stepy byłyby podobne, ponownie można zastosować ideę kompozycji i zacząć tworzyć stepy korzystając z interfejsów.

```
Skrypt aparatu Unity (1 odwołanie do zasobu) | Odwołania: 0
public class QuestGrabbableBox : GrabbableObject
{
    1 odwołanie
    private void CollectBox()
    {
        GameEventsManager.instance.questEvents.ChestCollected();
    }

    Odwołania: 4
    public override void Interact(GameObject interactedObject)
    {
        CollectBox();
        base.Interact(interactedObject);

    }
}
```

Skrzynia jest oczywiście GrabbableObjectem (bo można ją podnieść). To ona wywołuje metodę zawierającą event “onChestCollected”. Oprócz wywołania metody odpowiedzialnej za questa, skrzynia także korzysta z domyślnej metody interakcji z klasy bazowej. Napidpisując metodę Interact korzystając także ze słówka “base” korzystamy z domyślnego założenia metody, dodając jednak nową funkcjonalnosc niezbędną dla zadania.

Po podniesieniu skrzyni quest step został zakończony.

```
protected void FinishQuestStep()
{
    if (!isFinished)
    {
        isFinished = true;
        GameEventsManager.instance.questEvents.AdvanceQuest(questId);
        GameEventsManager.instance.questEvents.FinishStep(this);
    }
}
```

W tym momencie następuje ruszenie questa do przodu (AdvanceQuest)

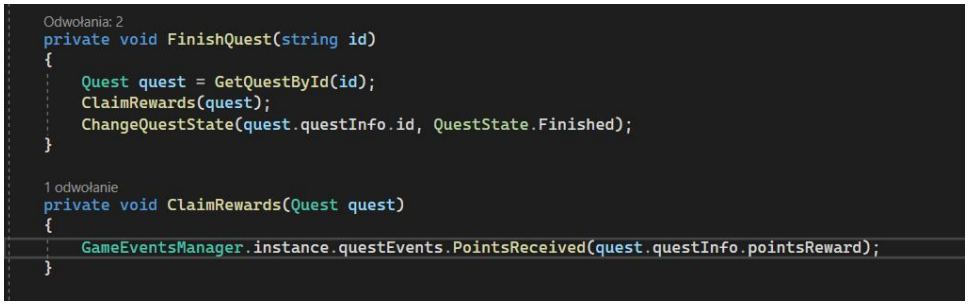
Ponownie QuestManager decyduje co się stanie dalej.

```
Odwołania: 2
private void AdvanceQuest(string id)
{
    Quest quest = GetQuestById(id);
    quest.MoveToNextStep();

    if (quest.IsStepExist())
    {
        quest.CreateQuestStep(this.transform);
    }
    else
    {
        ChangeQuestState(quest.questInfo.id, QuestState.CanFinish);
    }
}
```

Jeśli zadanie ma kolejny step to proces przechodzenia stepa powtarza się. W momencie, gdy jednak nie istnieje żaden kolejny step stan questa wskazuje na to, że staje się on gotowy do ukończenia. Jedyne co pozostało graczowi to wrócić do zleceniodawcy zadanie i oddać mu zadanie.

```
Odwołania: 2
public void StartOrFinishQuest(GameObject interactedObject)
{
    if (currentQuestState.Equals(QuestState.CanStart) && startPoint)
    {
        GameEventsManager.instance.questEvents.StartQuest(questId);
    }
    else if (currentQuestState.Equals(QuestState.CanFinish) && finishPoint)
    {
        GameEventsManager.instance.questEvents.FinishQuest(questId);
    }

    interactedObject.SetActive(!disableAfterInteract);
}
```

Z racji tego, że najpierw aktualizowany jest status questa, a dopiero potem wywołany “StartOrFinishQuest” to możliwe jest zakończenie questa od razu przy zakończeniu quest stepu. Z tego powodu, gdy wejdziemy w interakcje ze skrzynią, nie musimy szukać zleceniodawcy. Quest automatycznie jest zakończony.



```
Odwołania: 2
private void FinishQuest(string id)
{
    Quest quest = GetQuestById(id);
    ClaimRewards(quest);
    ChangeQuestState(quest.questInfo.id, QuestState.Finished);
}

1 odwołanie
private void ClaimRewards(Quest quest)
{
    GameEventsManager.instance.questEvents.PointsReceived(quest.questInfo.pointsReward);
}
```

Quest zakończony jest wtedy, gdy ma status “finished”. Gracz otrzymuje także nagrodę zawartą w scriptable objecie. W tym przypadku nagrodą będą punkty, które mogłyby także być warunkiem rozpoczęcia kolejnego zadania.