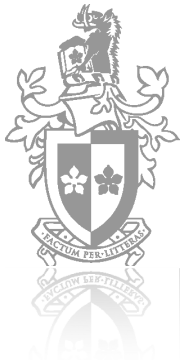


Object Oriented Programming

Swin-Adventure Case Study: C# Implementation Plan

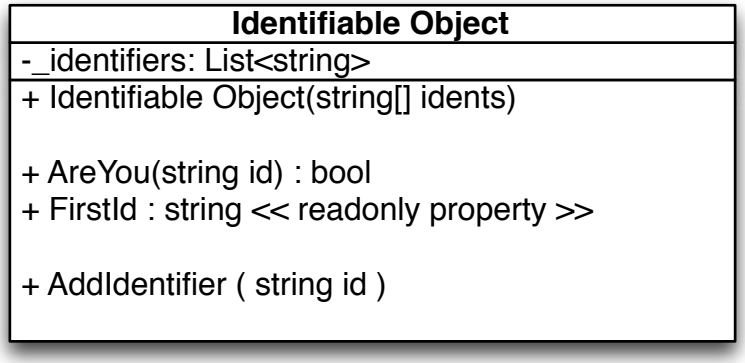
Outline

This document outlines the steps for implementing the core functionality of the Swin-Adventure program using the C# programming language.

Iteration 1: Identifiable Object

In iteration 1 you will create an Identifiable Object which will become the base class for many of the objects in the Swin-Adventure. Identifiable objects have a list of identifiers, and know if they are identified by a certain identifier.

The UML diagram for the Identifiable Object follows, and the unit tests to create are shown on the following page.



The player needs to be able to "identify" a number of things in the game. This includes commands the user will perform, items they will interact with, locations, paths, etc. The **Identifiable Object** role was created to encapsulate this functionality.

- Identifiable Object
  - The **constructor** adds identifiers to the Identifiable Object from the passed in array.
  - **Are You** checks if the passed in identifier is in the _identifiers
  - **First Id** returns the first identifier from _identifiers (or an empty string)
  - **Add Identifier** converts the identifier to lower case and stores it in _identifiers

Example Identifiable Object:

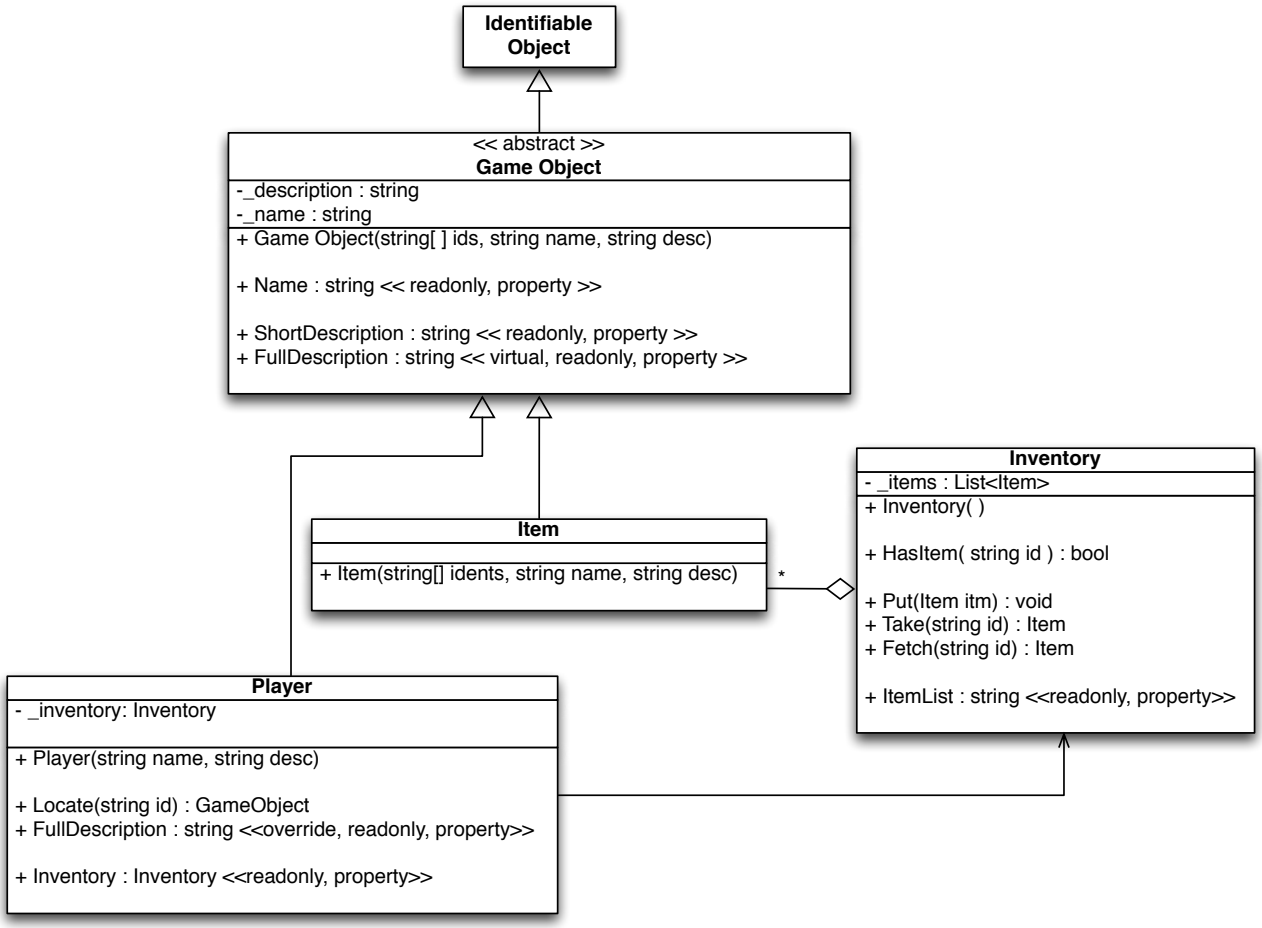
```
IdentifiableObject id =
    new IdentifiableObject( new string[] { "id1", "id2" } );
```

Use the following unit tests to create the Identifiable Object class and ensure that it is working successfully.

Identifiable Object Unit Tests	
Test Are You	Check that it responds "True" to the "Are You" message where the request matches one of the object's identifiers. eg. An Identifiable Object with identifiers “fred” and “bob” can be identified by (calling Are You) “fred” and “bob”.
Test Not Are You	Check that it responds "False" to the "Are You" message where the request **does not** match one of the object's identifiers. eg. An Identifiable Object with identifiers “fred” and “bob” can NOT be identified by (calling Are You) “wilma” or “boby”.
Test Case Sensitive	Check that it responds "True" to the "Are You" message where the request matches one of the object's identifiers where there is a mismatch in case. eg. An Identifiable Object with identifiers “fred” and “bob” can be identified by (calling Are You) “FRED” and “bOB”.
Test First ID	Check that the first id returns the first identifier in the list of identifiers. eg. An Identifiable Object with identifiers “fred” and “bob” has "fred" as its First ID
Test Add ID	Check that you can add identifiers to the object. eg. An Identifiable Object with identifiers “fred” and “bob” can have "wilma" added and then be identified by (calling Are You) with “fred”, “bob”, and "wilma".

Iteration 2 - Players, Items, and Inventory

The goal of this iteration will be to create the Player, Item, and Inventory classes and the ability to locate things within these objects. From this it should be possible to add items into the Player's inventory and then get the player to locate the item for us.



Notes:

- The Player constructor will call the GameObject constructor and pass up identifiers for "me" and "inventory".

```
public Player(string name, string desc) :
    base( new string[] { "me", "inventory"} , name, desc)
{
    ...
}
```

This iteration adds several classes that help create a number of key abstractions for the game. The **Game Object** class will be used to represent anything that the player can interact with.

- Game Object - "anything" the player can interact with
  - Name - this is a short textual description of the game object
  - Description - a longer textual description of the game object
  - Short Description - returns a short description made up of the name and the first id of the game object. This is used when referring to an object, rather than directly examining the object.
  - Long Description - By default this is just the description, but it will be changed by child classes to include related items.

Example from requirements document:

```
Command -> inventory
You are Fred the mighty programmer.
You are carrying
    a shovel (shovel)
    a bronze sword (sword)
    a small computer (pc)
```

The inventory command is the same as "look at me", so this shows the Full Description of the Player. It includes the short description if items the player is carrying, for exam "a shovel" is the name of an item the player is carrying. The first id of this item is "shovel", so its Short Description is "a shovel (shovel)".

In the design the **Item** class is a kind of **Game Object**. This is an abstraction of the "Items" the player can interact with in the game. Example item:

```
Item shovel = new Item( new String[] {"shovel", "spade" },
                        "a shovel",
                        "This is a might fine ..." );
```

Player is also a kind of **Game Object**. This will be a object through which the player will interact with the game world.

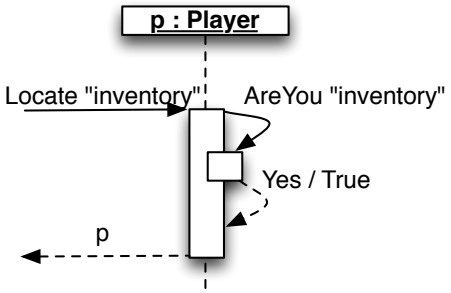
- Player - the players avatar in the game world
  - An Inventory object is used to manage the Player's items
  - Full Description is overridden to include details of the items in the player's inventory.
  - Locate "finds" a GameObject somewhere around the player. At this stage that includes the player themselves, or an item the player has in their inventory. See next page...

A number of GameObjects will need to contain Items. This functionality is encapsulated in the **Inventory** class. This provides a managed list of items.

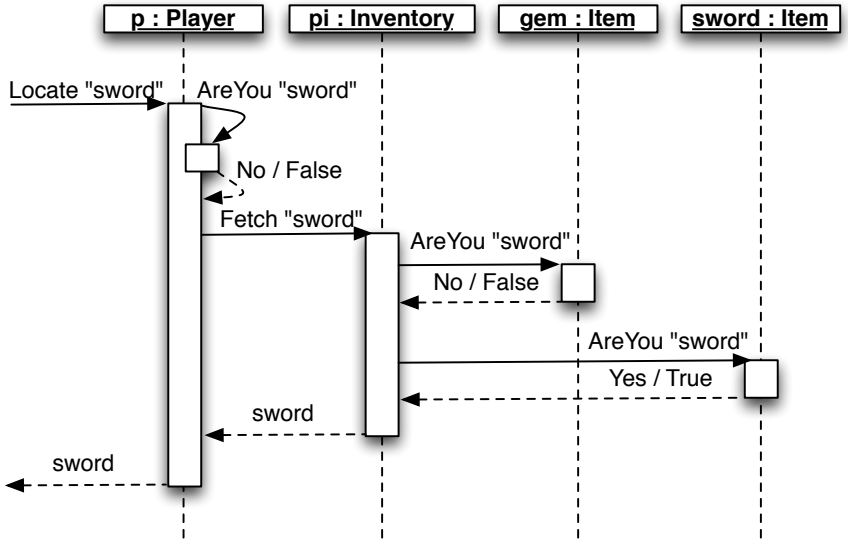
- Inventory - a managed collection of items
  - Items can be added using Put, or removed by id using Take
  - Fetch locates an item by id (using AreYou) and returns it

The following UML sequence diagrams shows the sequence of messages involved in locating the player and their items.

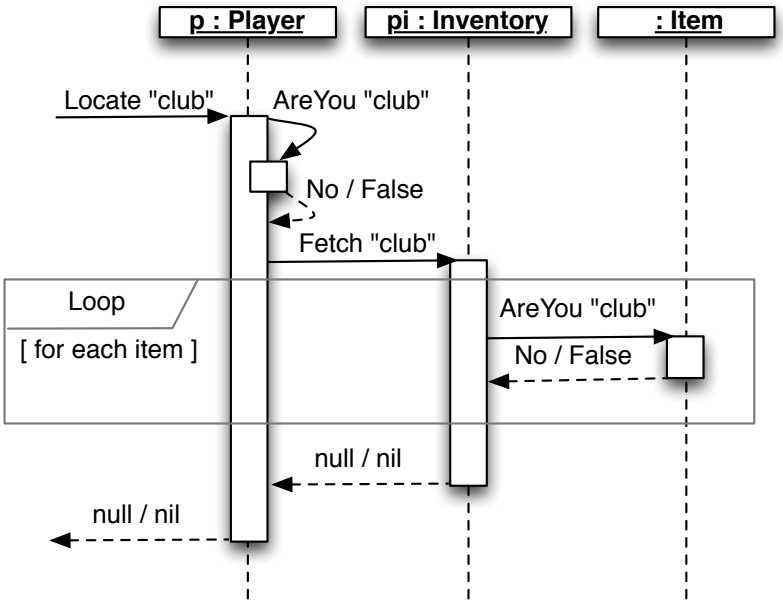
The player can "locate" themselves.



The player can locate items in their inventory. Note: *pi* is the player's inventory object.



When there are no items that match then null/nil is returned.



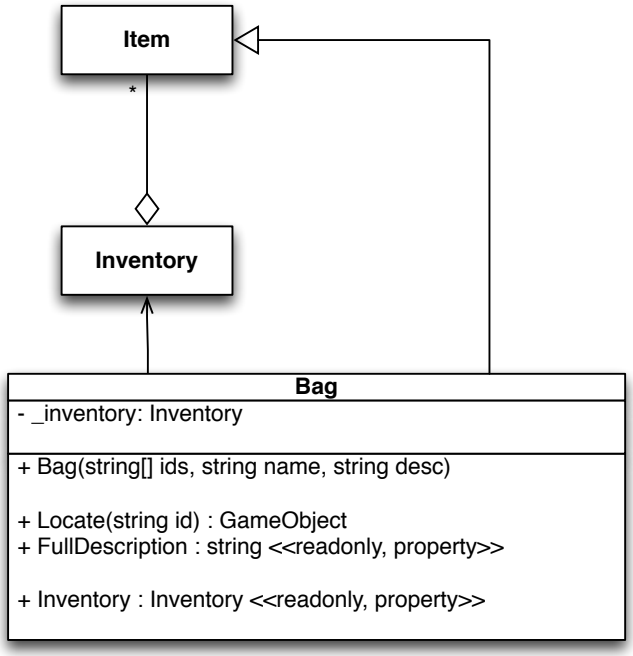
Item Unit Tests	
Test Item is Identifiable	The item responds correctly to "Are You" requests based on the identifiers it is created with.
Test Short Description	The game object's short description returns the string "a name (first id)" eg: a bronze sword (sword)
Test Full Description	Returns the item's description.

Inventory Unit Tests	
Test Find Item	The Inventory has items that are put in it.
Test No Item Find	The Inventory does not have items it does not contain.
Test Fetch Item	Returns items it has, and the item remains in the inventory.
Test Take Item	Returns the item, and the item is no longer in the inventory.
Test Item List	Returns a list of strings with one row per item. The rows contain tab indented short descriptions of the items in the Inventory.

Player Unit Tests	
Test Player is Identifiable	The player responds correctly to "Are You" requests based on its default identifiers (me and inventory).
Test Player Locates Items	The player can locate items in its inventory, this returns items the player has and the item remains in the player's inventory.
Test Player Locates itself	The player returns itself if asked to locate "me" or "inventory".
Test Player Locates nothing	The player returns a null/nil object if asked to locate something it does not have.
Test Player Full Description	The player's full description contains the text "You are carrying:" and the short descriptions of the items the player has (from its inventory's item list)

Iteration 3 - Bags

In this iteration you will add a Bag class to make it possible to have items that contain other items.

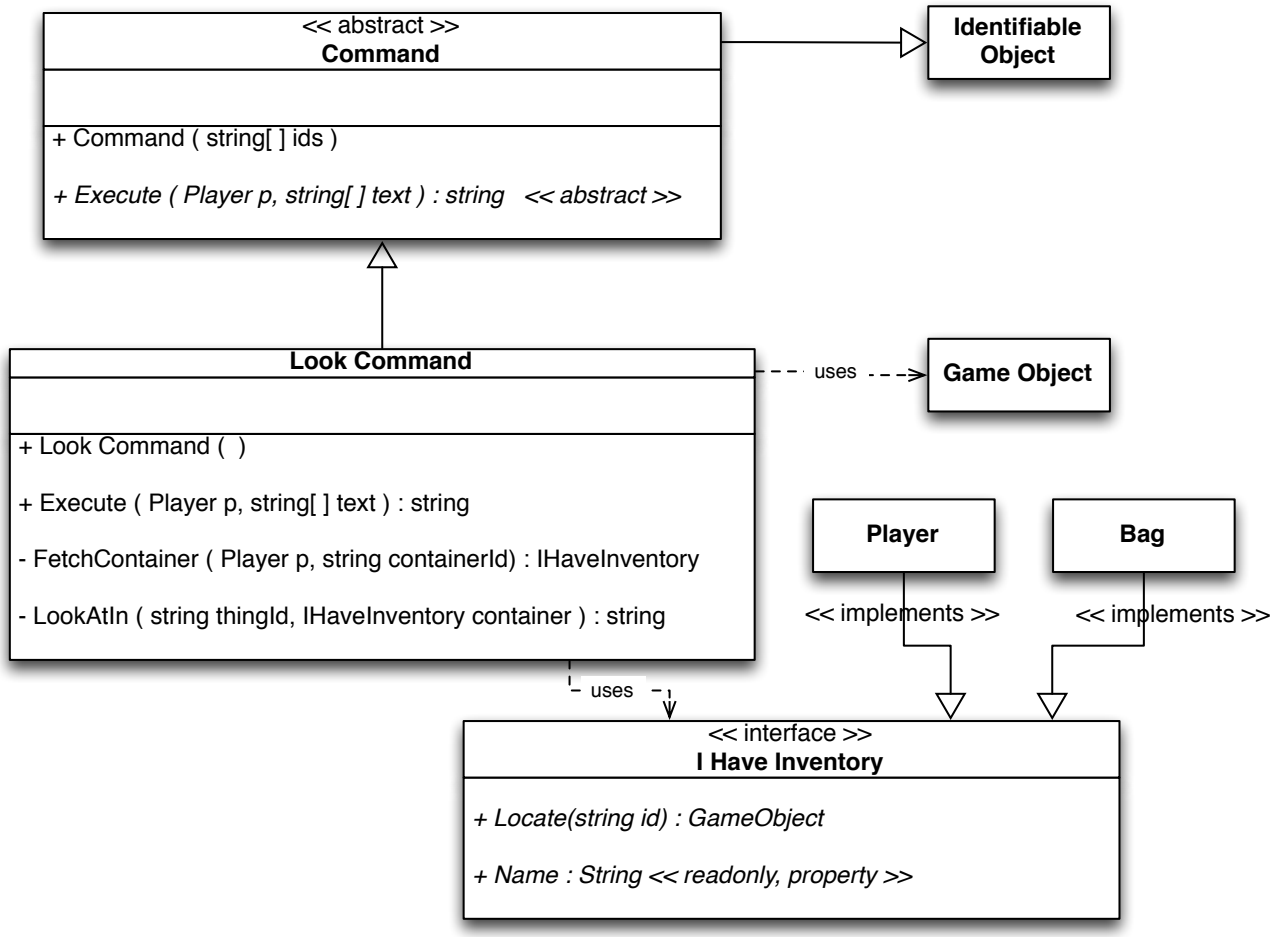


The Bag abstraction is a special kind of item, one that contains other items in its own Inventory. This is a version of the **composite pattern**, which allows flexible arrangements of bags and items, for example a bag to contain another bag.

Bag Unit Tests	
Test Bag Locates Items	You can add items to the Bag, and locate the items in its inventory, this returns items the bag has and the item remains in the bag's inventory.
Test Bag Locates itself	The bag returns itself if asked to locate one of its identifiers.
Test Bag Locates nothing	The bag returns a null/nil object if asked to locate something it does not have.
Test Bag Full Description	The bag's full description contains the text "In the <name> you can see:" and the short descriptions of the items the bag contains (from its inventory's item list)
Test Bag in Bag	Create two Bag objects (b1, b2), put b2 in b1's inventory. Test that b1 can locate b2. Test that b1 can locate other items in b1. Test that b1 **can not** locate items in b2.

Iteration 4 - Looking

In this iteration you will add the first of the commands, the look command. This will give you sufficient code to create a small application where the user can look at items they have.



As there will be a number of Commands, an abstract **Command** class has also been added. This class inherits from **Identifiable Object**, as each of the commands needs to be identifiable. When data is entered by the user, each Command object will be asked “Are You” and the first work of the command. For example, with “look at pen” each Command would be asked “Are You ‘look’ ” to locate the Command to process the text. As a result the Look Command should be identified by “look”.

The Look Command will handle the instructions like:

```
look at pen in bag
look at bag in inventory
look at pen
look at bag
```

To handle this the Look Command will need to perform a set of action:

1. Locate the "container" where the item resides. For example the *bag* in "look at pen in bag", or the *player* in "look at pen".
   - Return "I cannot find the <<container text>>" if the container cannot be found. For example, "look at pen in bag" may return "I cannot find the bag".
2. Locate the item from the "container"
   - Return "I cannot find the <<item text>> in the <<container name>>" if the item cannot be found. For example, "look at pen in bag" may return "I cannot find the pen in the small bag" (in this case "small bag" is the name of the Bag container).
3. Return the full description of the Game Object found.

For this to work the code needs a way of treating Bags and Players in the same way: as objects that container other items. To handle this the **I Have Inventory** interface/protocol has been added. Objects that implement this must be able to perform the following tasks for the Look Command:

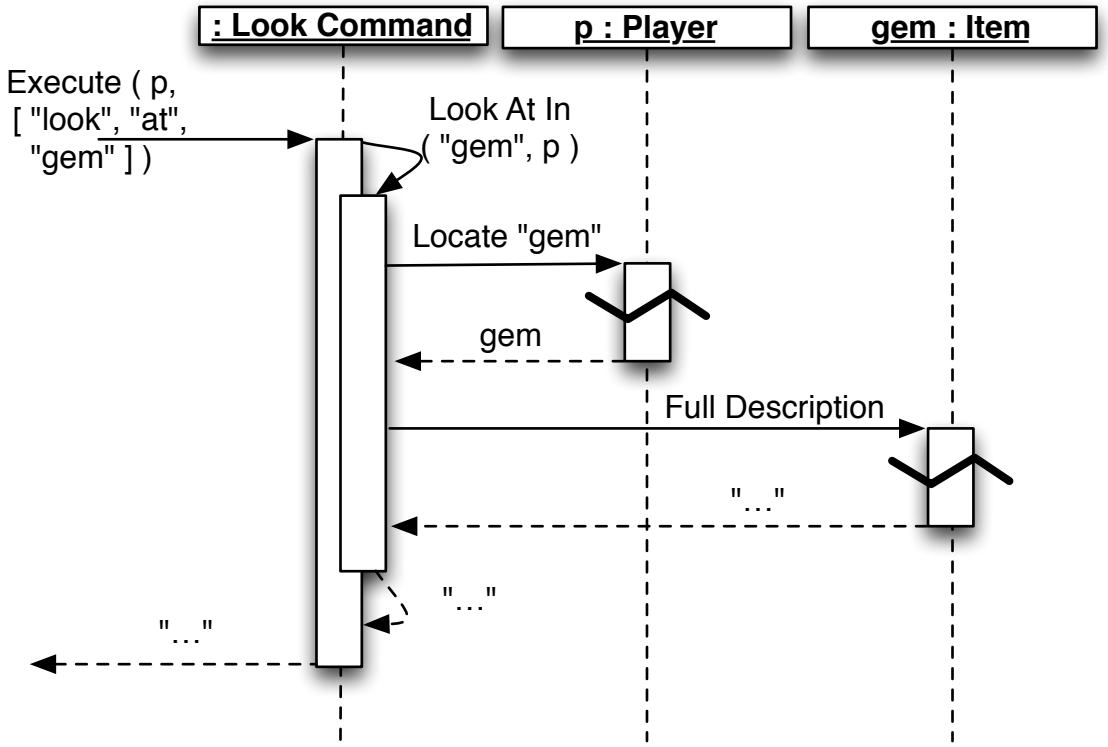
- **Locate** and item given its id. Once the container is located from the Player, it is then asked to locate the item from the text id the user entered.
- The container needs to be able to return its **Name**. This can then be used when the item cannot be found in the container.

The command input will be supplied as a list of individual words. For example: "look at pen in bag" will be passed in as the list ["look", "at", "pen", "in", "bag"]. The conversion of the text from a single string to a list will be handled elsewhere.

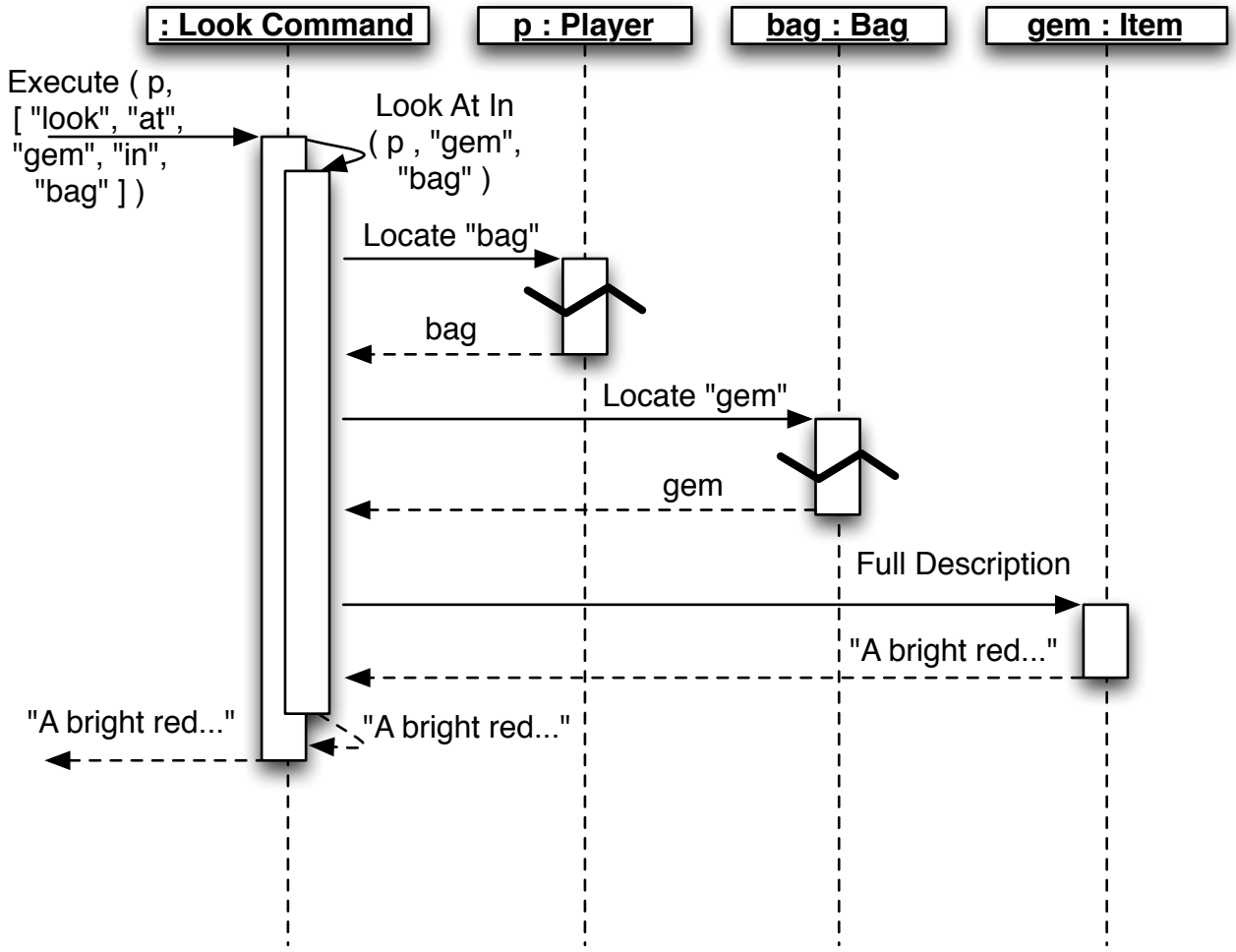
Processing the array will be performed as follows:

1. There must be either 3 or 5 elements in the array, otherwise return "I don't know how to look like that"
2. The first word must be "look", otherwise return "Error in look input"
3. The second word must be "at", otherwise return "What do you want to look at?"
4. If there are 5 elements, then the 4th word must be "in", otherwise return "What do you want to look in?"
5. If there are 3 elements, the container is the player
6. If there are 5 elements, then the container id is the 5th word
   1. Call FetchContainer, and have this method retrieve the container from the Player.
7. The item id is the 3rd word
8. Perform the look at in, with the container and the item id

The following shows the sequence of messages involved in performing the command "look at gem". In this case the container is the player themselves, though the logic is the same as when it is another object that matches the I Have Inventory interface/protocol.



The following sequence diagrams shows the command "look at gem in bag". When the command is "look at x in y" then the **Locate Container** method asks the player to locate "y", and returns this as the container to locate "x".



Notes:

- You will need to cast the `GameObject` to `IHaveInventory` use the following to perform a safe type cast. This will set container to null if obj is not an object that implements the I Have Inventory interface/protocol.

```
container = obj as IHaveInventory;
```

Look Command Unit Tests	
Test Look At Me	Returns players description when looking at "inventory"
Test Look At Gem	Returns the gems description when looking at a gem in the player's inventory.
Test Look At Unk	Returns "I can't find the gem" when the the player does not have a gem in their inventory.
Test Look At Gem In Me	Returns the gem's description when looking at a gem in the player's inventory. "look at gem in inventory"
Test Look At Gem in Bag	Returns the gems description when looking at a gem in a bag that is in the player's inventory.
Test Look at Gem in No Bag	Returns "I can't find the bag" when the the player does not have a bag in their inventory.
Test Look at No Gem in Bag	Returns "I can't find the gem" when the the bag does not have a gem in its inventory.
Test Invalid Look	Test look options to check all error conditions. For example: “look around”, or “hello”, “look at a at b”, etc.

Iteration 5 - Create a Console Application

For the application:

- Get the player's name and description from the user, and use these details to create a Player object.
- Create two items and add them to the the player's inventory
- Create a bag and add it to the player's inventory
- Create another item and add it to the bag
- Loop reading commands from the user, and getting the look command to execute them.

Iteration 6 - Adding Locations

Use the following information to help you design the additions necessary to add locations to the Swin-Adventure.

- Locations:
  - Will need to be identifiable and have a name, and description.
  - Can contain items.
- Players are in a location.
- Players "locate" items by checking three things (in order):
  - First checking if they are what is to be located (locate "inventory")
  - Second, checking if they have what is being located (_inventory fetch "gem")
  - Lastly, checking if the item can be located where they are (_location, locate "gem")
- This will change the look command to also include "look" to look at the player's location.

Here are some hints for things you will need to test for:

- Locations can identify themselves
- Locations can locate items they have
- Players can locate items in their location

Tasks:

- Draw a UML class diagram to show what needs to be added
- Draw a UML sequence diagram to explain how locate works in the Player, with the newly added Location aspect to the search.
- Implement the unit tests, and features to support them.

Iteration 7 - Paths and Moving

Implement Path, Direction, and the Move Command.

Notes:

- Have the Path objects move the player to the new location. This will allow for flexibility like lockable paths etc.
- Make Path's identifiable. The identifiers indicate the direction, and can be used to locate the path from the location.
- The Move Command is identified by the words "move", "go", "head", "leave", ...

Here are some hints for things you will need to test for:

- Path can move player to the Path's destination
- You can get a Path from a Location given one of the path's identifiers
- Players can leave a location, when given a valid path identifier
- Players remain in the same location when they leave with an invalid path identifier

Tasks:

- Draw a UML class diagram to show what needs to be added
- Draw a sequence diagram to explore how moving will work. For example: execute "move north" is sent to a Move Command.
- Implement the unit tests, and features to support them.

Iteration 8 - Command Processor

The command processor will contain a list of Command objects (one of each kind of Command). This can be used to **execute** the user's commands. When execute "a command" is given to the Command Processor, it searches for a command that is identified by the first word, then tells it to execute the text. For example, execute "move north" the Command Processor will look for the Command that is identified by the "move" id and then tell it to execute ["move", "north"] for the player.

Tasks:

- Draw a UML class diagram to show what needs to be added
- Draw a sequence diagram to explore how executing a command works (ignoring the internal details of the Commands). For example "look at gem in bag" being sent to the Command Processor.
- Implement the unit tests, and features to support them.
- Convert the application to use the Command Processor

Iteration 9+

Plan out iterations for:

- Transfer Command to perform put and take
- Maze loading from text file
- Quit Command
- GUI anyone? Creating a simple GUI using Windows Forms in C# is not overly challenging, even without the visual design tools from Visual Studio.