Faculty of Information and Communication Technologies

Object Oriented Programming

Swin-Adventure Case Study: C++ Implementation Plan

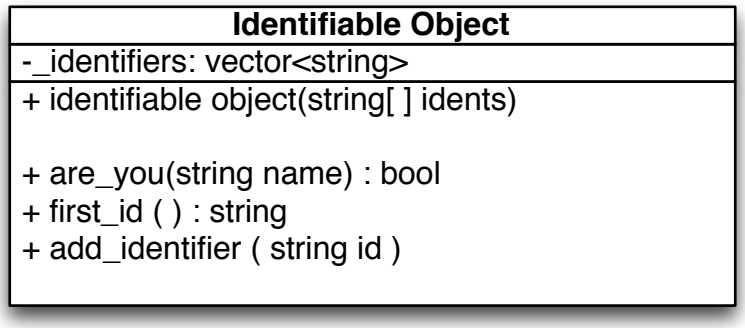
Outline

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.



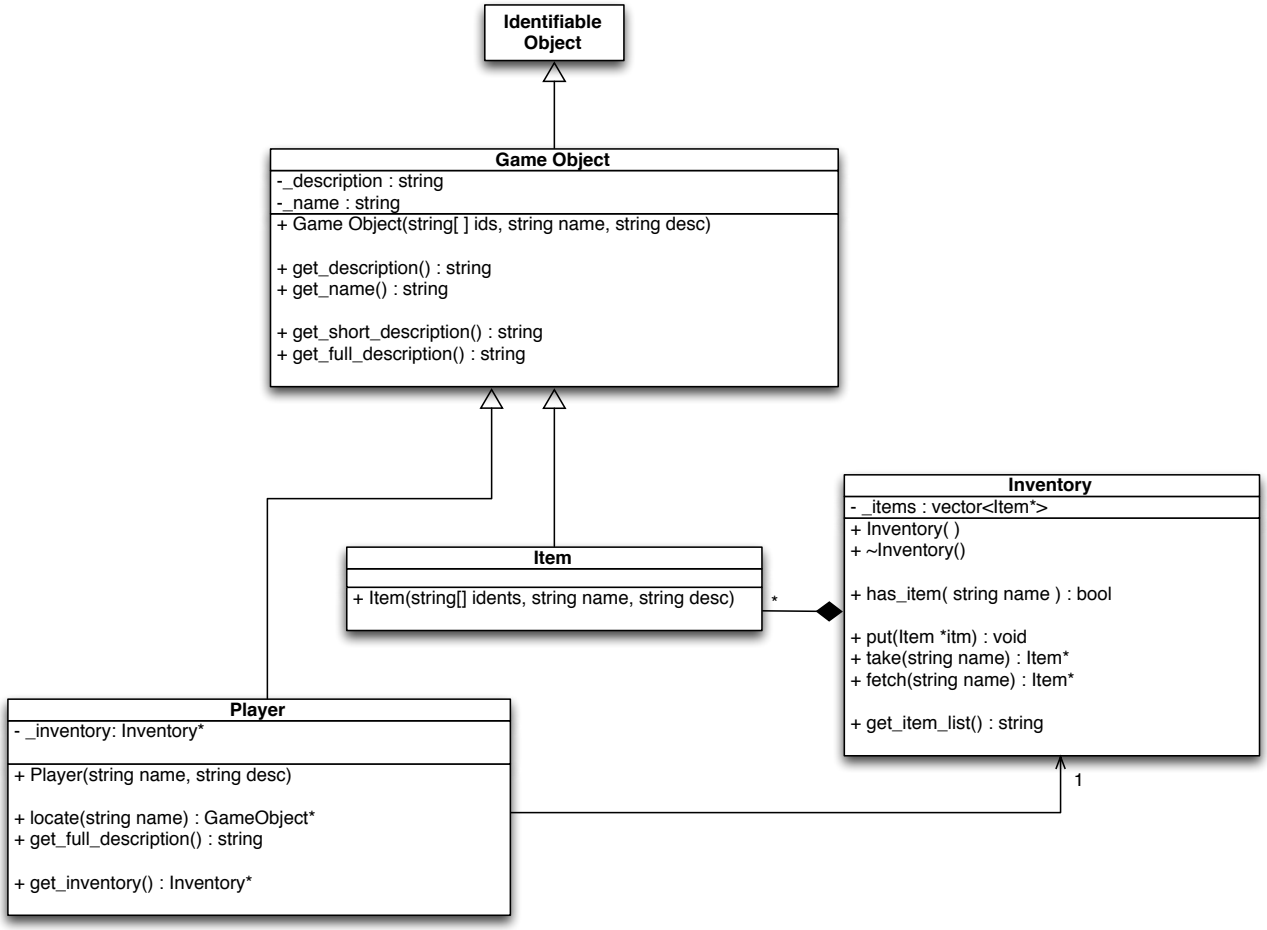
Use the following unit tests to create the Identifiable Object class and ensure that it is working successfully.

Identifiable Object Unit Tests	
Test Creation	Check you can create an identifiable object with a list of identifiers.
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

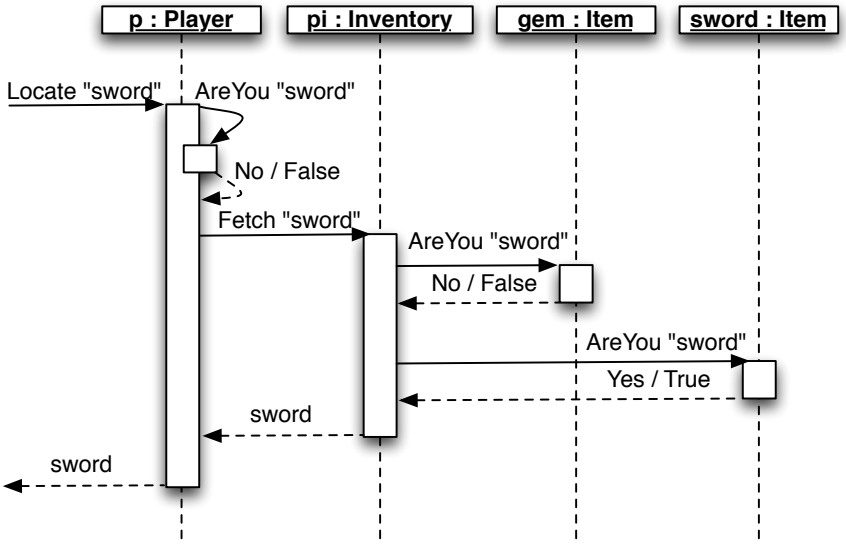
The goal of this iteration will be to create the Player, Item, and Inventory classes and the ability to locate things within these objects. From this it should be possible to add items into the Player's inventory and then get the player to locate the item for us.

The UML diagram for the Identifiable Object follows, and the unit tests follow.

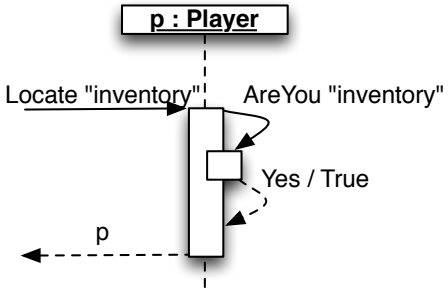


The following UML sequence diagrams shows the sequence of messages involved in locating the player and their items.

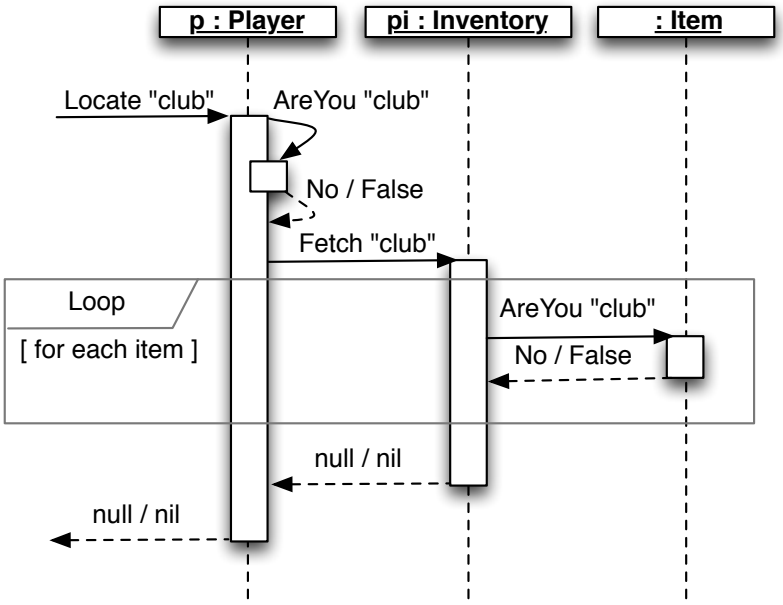
Note: *pi* is the player's inventory object.



The player can also "locate" themselves.



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



Game Object Unit Tests	
Test Name	The game object remembers the name it is given when it is created.
Test Description	The game object remembers the description it is given when created.
Test Game Object is Identifiable	The game object 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 if)" eg: a bronze sword (sword)
Test Full Description	Returns the items description.

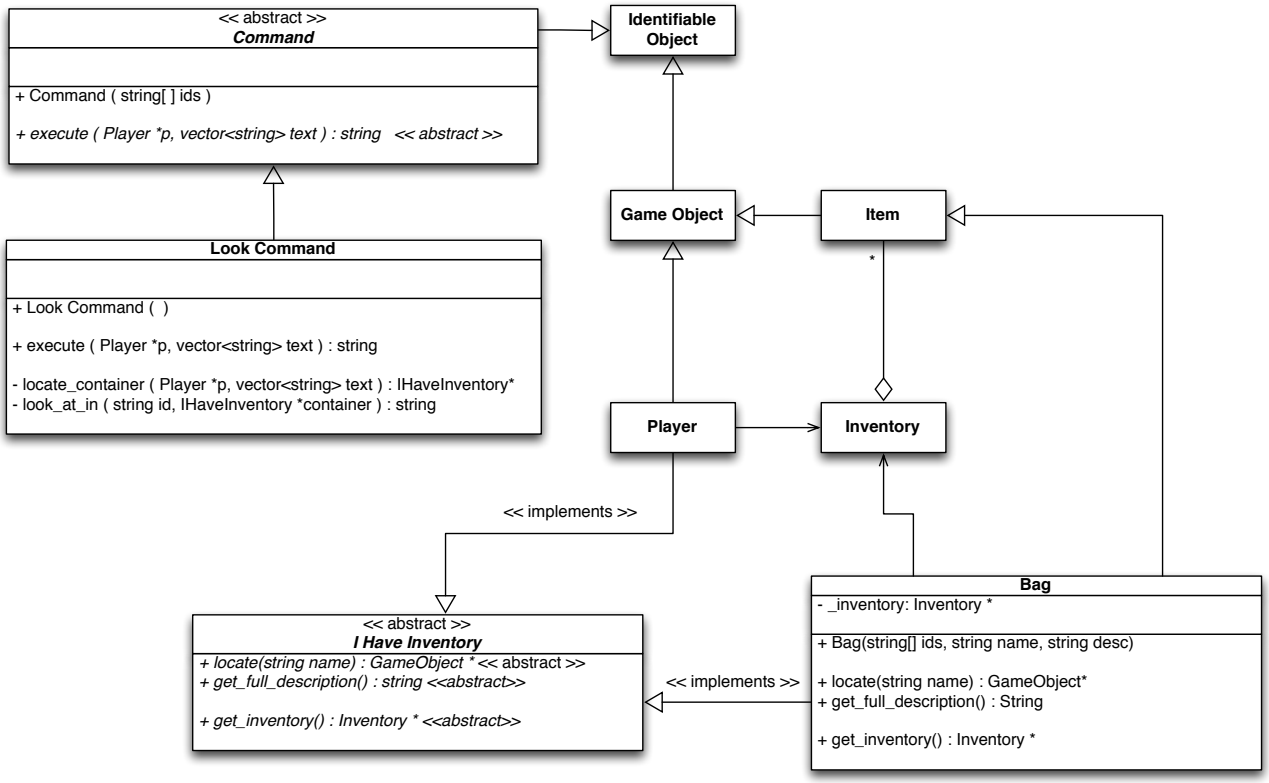
Item Unit Tests	
Test Item is Identifiable	The item responds correctly to "Are You" requests based on the identifiers it is created with.

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 added.

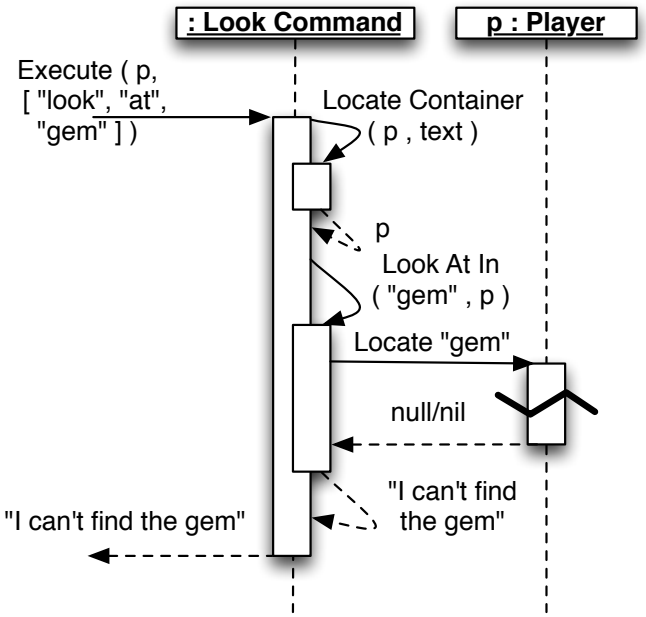
Player Unit Tests	
Test Player is Identifiable	The player responds correctly to "Are You" requests based on its default identifiers (me and inventory).
Test Player Locates Items	The player can locate items in its inventory, this returns items the player has and the item remains in the player's inventory.
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 and Looking

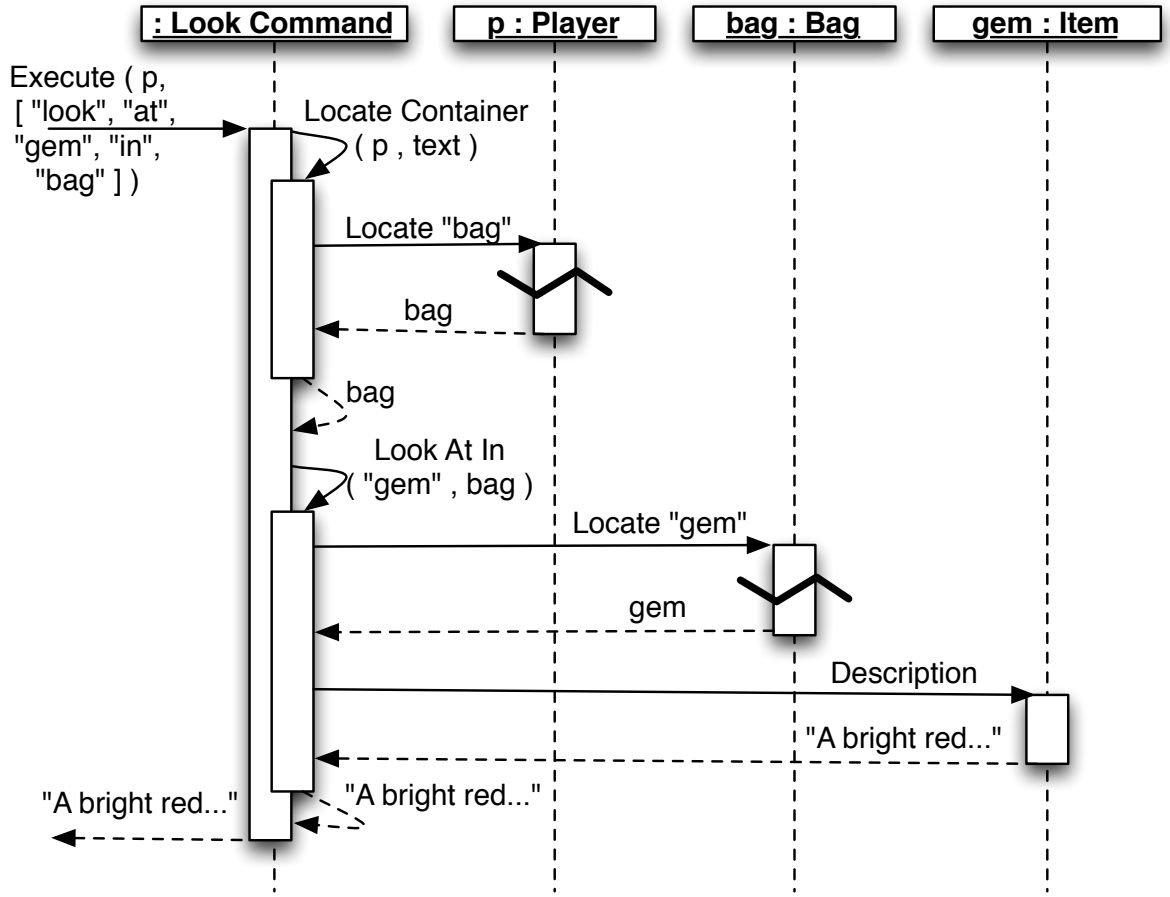
In this iteration you will add the Bag class, and add the first of the commands. This will give you sufficient code to create a small application where the user can look at items they have.



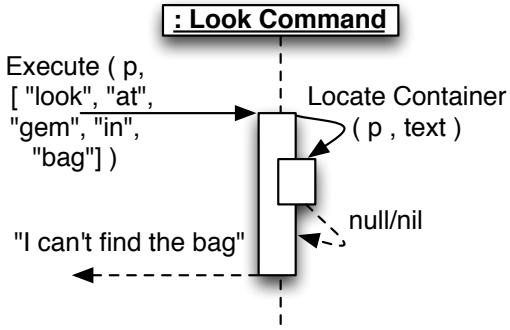
The following shows the sequence of messages involved in performing the command "look at gem". In this case the player does not have a gem in their inventory. The the command is "look at x", then the **Locate Container** method returns the player object.



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".



A similar "can't find" message is returned when the player does not have the "bag".



Notes:

- Look Command is identified by "look", "examine", "inspect", ...
- You will need to search for functionality from the library to split the input string into an array of words (based on " ").

Bag Unit Tests	
Test Bag Locates Items	The bag can locate 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 "bag" if this is one of the bag's identifiers (for example).
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 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)

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.

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 4 - 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, fetch "gem")

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
- The move command can process execute (p , "move north")

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 5 - 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 6 - 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 7 - GUI

Design and implement a simple GUI for the game.

Iteration 8+

Plan out iterations for:

- Transfer Command to perform put and take
- Maze loading from text file
- Quit Command