**Design Patterns**

**Project Report**

**Title:** Emulation of network components for tracing ping packets and checking for correctness

**Motivation:** The basic setup of the network, its components, and their features is based on Cisco Packet Tracer. Basic functionality of Cisco Packet Tracer like pinging, tracing of the ping andpre-checks for network components is also implemented in this project

**Features:**

* A network composed of any of the mentioned components can be setup
* Pinging functionality from one network node to another
* The success or failure of this ping
* The trace of the path the ping takes
* Basic Format checks implemented for all attributes
* A comfortable Command based UI for easy setup

**Detailed Description of Components:**

* **PC:**
  + **Name:** To ensure easy identification and referring of components for user.
  + **Gateway:** A network node that connects two dissimilar networks using different protocols together. The default gateway always resides in the same subnet as the end device IP. The gateway can really be any unique address within the subnet itself, but most network administrators designate the first number of the subnet as the gateway.
  + **SubnetMask:** The subnet mask is used to subdivide a network into smaller, more manageable chunks. In order for a sending device to transmit data to a receiving device, the sender needs to know where the destination is. The destination will either be on the same subnetwork as the source, or on some other subnetwork.
  + **IPAddress:** Every device connected to a network must have an unique IP address to differentiate it from the others. It consists of four segments called octets that are separated by a period. The numbers within each octet range between 0 and 255.
* **Hub:**
  + **Name:** To ensure easy identification and referring of components for user
* **Router:**
  + **Name:** To ensure easy identification and referring of components for user
  + **Routes:** Represents the routing table. A routing table contains the information necessary to forward a packet along the best path toward its destination. Each packet contains information about its origin and destination. A basic routing table includes the following information:
    - **Destination:** The IP address of the packet's final destination
    - **Next hop:** The IP address to which the packet is forwarded
    - **Subnet Mask:** The associated subnet mask
  + **Fast Ethernet Port:** Represents the 4 NIC interfaces on the router. It has the following information associated with it:
    - **MAC Address:** a unique identifier assigned to network interfaces for communications at the data link layer of a network segment
    - **IP Address:** IP Address that represents this port
    - **Subnet Mask:** The associated subnet mask
* **Links:** Represented using the ‘adjMat’ data structure that is explained later.

**Adjacency List for link storage**

To store the linkage between various components, three adjacency matrices were setup for efficient computation. Each matrix stored the links for each node in the network. For faster computation and to avoid redundant conversions and computations, the above data was stored in three matrices, each of which was indexed by one the three attributes mentioned below:

* Name of the node
* IP of the node
* The node itself

**Creation of Components using Builder pattern supplemented with Abstract Factory Pattern**

**Builder Pattern:**

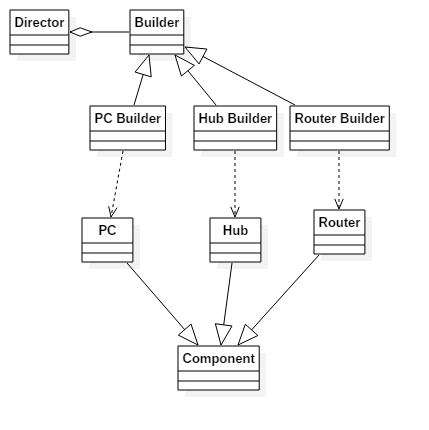
* **Intent:** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
* **General Methodology:** The "director" invokes "builder" services as it interprets the external format. The "builder" creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the "builder".

**Abstract Factory Pattern:**

* **Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
* **General Methodology:** Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

**How did we use them?**

The Abstract Builder Class represents the general builder that would build a networking component. No default implementations are provided. Concrete Builders are designed for the creation of individual components: Hubs, PCs and Routers. Now, this is where the design pattern is inspired by Abstract Factories. The system is modelled to ensure that a specific networking component implements operations from its own family of operations only. Each concrete builder selectively implements only the methods relevant to the component associated with it. This is to ensure that even if a function, that is not relevant to the builder under consideration, is invoked, it will be ignored and no action will be taken pertaining to it. Due to this flexibility, we are able to have a common director class that directs the building process for all components and conditional statements are completely avoided. This design ensures that the builders implement the methods relevant to them only, but without the use of conditional statements or duplicate code. Hence, we essentially implement the Builder pattern with slight modifications to it that are inspired from the Abstract Factory pattern. Apart from the builder classes, classes are also implemented to represent the individual components: Hub, PC and Router. These classes implement the getters and setters relevant to each component and inherit from the common abstract base class - Component

****

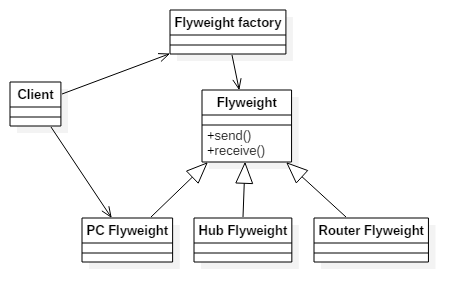
**Intrinsic functionality of Components using Flyweight Pattern:**

**Flyweight Pattern:**

* **Intent:** To use sharing to support large number of fine-grained objects efficiently, and to replace heavy-weight components with light-weight alternatives
* **General Methodology:** A “flyweight” class is created to hold the intrinsic properties and functionality common to a set of objects. A hierarchy of flyweight classes may be necessary to distinguish between types of objects. A Flyweight Factory is maintained holding a list of instances of each flyweight class. Flyweight Factory is queried with a parameter to specify which flyweight object is required, and the appropriate flyweight is returned. A class using flyweight pattern queries the Flyweight Factory with an appropriate parameter, and store a reference to the shared flyweight object.

**How did we use it?**

A common Flyweight base class defining an interface for all flyweight classes is defined, to facilitate flexibility and uniformness. Flyweight classes, each extending the common Flyweight base class, are defined for each type of component, namely the PCs, Hubs and Routers. These flyweight classes contain the intrinsic functionality of each type of device, namely their behavior on receiving a ping, and on sending a ping. Flyweight Factory keeps an instance of each flyweight class, namely PCFlyweight, HubFlyweight, and RouterFlyweight. When an object of one of the device classes is created, Flyweight Factory is queried, and a reference to the appropriate flyweight object is returned. Thus, the flyweight objects containing intrinsic functionality are shared between all objects containing them.



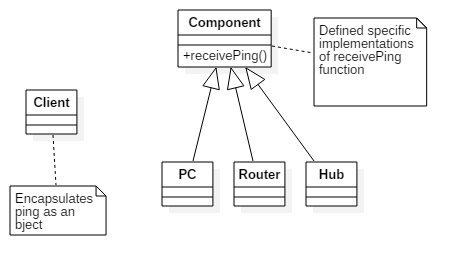
**Ping functionality using concepts of Command Pattern:**

**Command Pattern:**

* **Intent:** Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations. Invocations of methods on objects are promoted to full objects.
* **General Methodology:** The client that creates a command may not be the client executing it. This separation provides flexibility in the sequencing of command, and in the flexibility of which client handles the command. The command is promoted to an object, and can be passed, shared, or manipulated like any object. All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual execute() method whenever the client requires the object's "service".

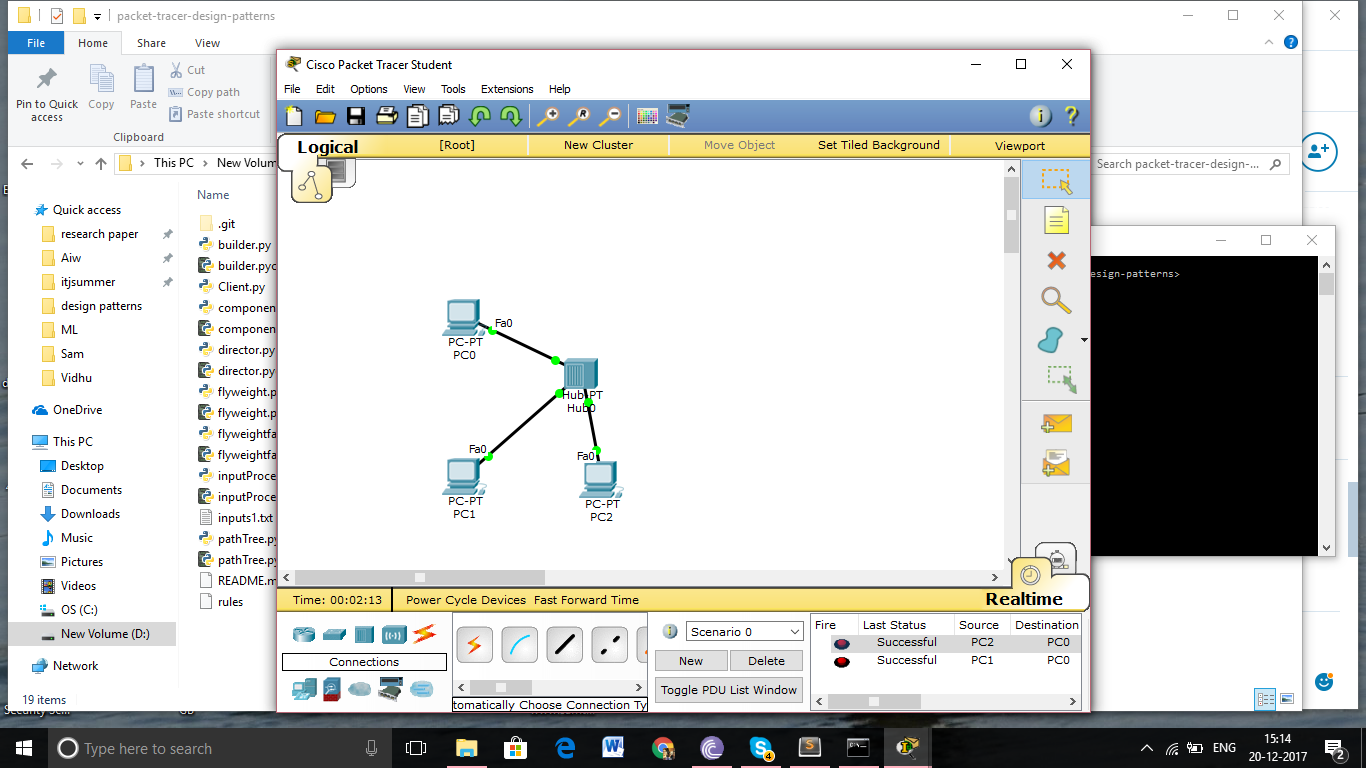
**How did we use it?**

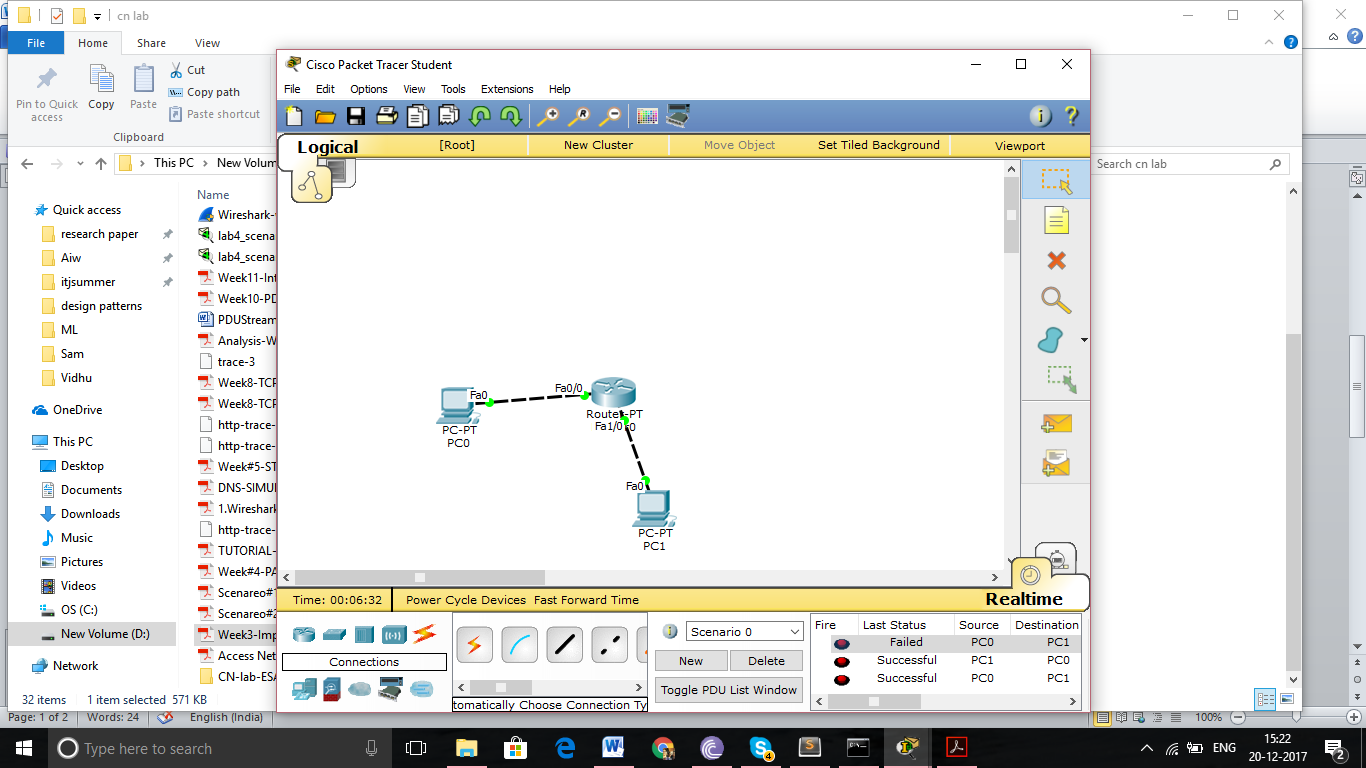
When networks are constructed, users are given the opportunity to test their networks using pings, from one PC to another. Pings are specified by source IP address and destination IP address, and these ping requests are encapsulated into objects, consisting of source IP address, destination IP address, and a list of devices already visited during the course of the ping’s traversal through the network. A log of visited devices is maintained through the course of the ping’s traversal, and each prospective path taken is plotted. Devices are treated as black boxes, and the client only invokes the send() and receive() functions when the ping forwarding or reception services are required respectively.

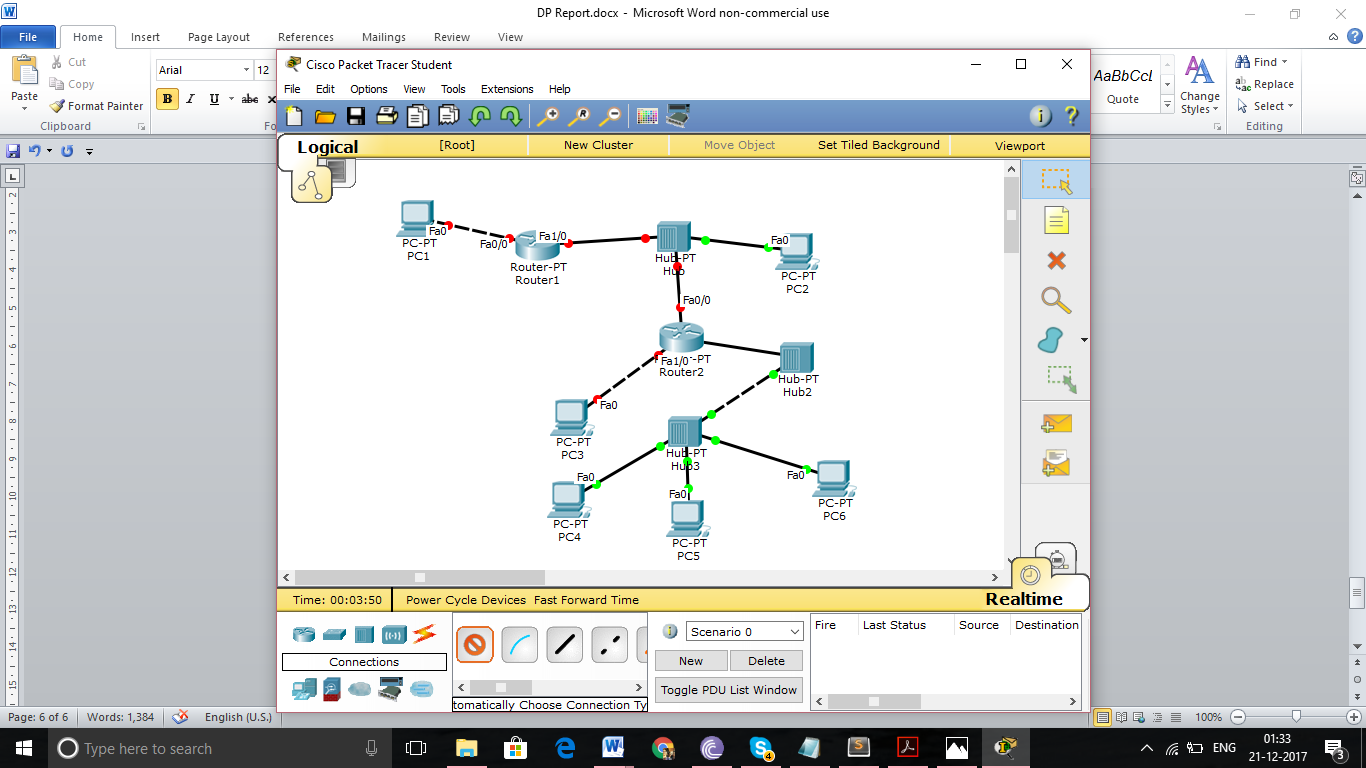


**Implementation Language:** Python

**Demo Setups:**







**Team Details:**

* Srinidhi Sridhar, 01FB14ECS252
* Vidhu Malik, 01FB14ECS283