S: একটি ক্লাস শুধুমাত্র একটি নির্দিষ্ট কাজের জন্য নিয়োজিত থাকবে।  
O: কোড পরিবর্তন না করে নতুন ফিচার যোগ করা যাবে।  
L: সাবক্লাসগুলো মূল ক্লাসের মতোই কাজ করবে।  
I: বড় ইন্টারফেসের বদলে ছোট ছোট ইন্টারফেসে ভাগ করে কাজ করবে।  
D: নির্দিষ্ট ক্লাসের উপর নির্ভর না করে ইন্টারফেসের উপর নির্ভর করবে।

আজকে আমরা কথা বলি SOLID Principle নিয়ে। SOLID Principle আসলে জিনিসটা কি?? SOLID হলো সফটওয়্যার ডেভেলপমেন্টে Object Oriented Design (OOD) এর পাঁচটি নিয়মনীতির সমষ্টী । এইগুলো কোডকে আরো বেশি রিইউজেবল এবং মেইনটেনেবল করতে সাহায্য করে । চলুন আমরা পাঁচটি নীতি সম্পর্কে বিস্তারিত আলোচনা করিঃ   
  
  
S – Single Responsibility Principle:  এই নীতি অনুসারে প্রতিটি ক্লাস বা মডিউল শুধুমাত্র একটি দায়িত্ব পালন করবে। কেননা একটি ক্লাসে একাধিক কাজ করে হলে কোড জটিল হয়ে যায় এবং তা পরিবর্তন করাও কষ্টসাধ্যেমনঃ  ধরুন একটি ক্লাস ইউজার রেজিস্ট্রেশনের কাজ করছে সেই ক্লাস লগইন বা অন্য কোন কাজ করবে না । অর্থাৎ প্রতিটি ক্লাস একটি নির্দিষ্ট কাজ সম্পূর্ণ করবে ।   
  
O – Open-Closed Principle :  একটি ক্লাস বা ফাংশনকে নতুন ফিচার যোগ করা বা  এক্সটেনশন করার জন্যে ওপেন থাকবে কিন্তু পরিবর্তন করার জন্য ক্লোজড থাকবে। যেমনঃ ধরুন আপনার ইউজার নামে একটা ক্লাস আছে এখন আপনি লগইন, রেজিস্ট্রেশনের কাজ ঐ ইউজার ক্লাসে না করে ঐ ইউজার ক্লাসকে ইনহেরিট করে লগইন, রেজিস্ট্রেশন বা অন্যান্য কাজ করবেন । অর্থাৎ ইনহেরিটেন্সের মাধ্যমে নতুন ফিচার যোগ করা।   
  
L – Liskov Substitution Principle :   সাবক্লাসকে প্যারেন্ট ক্লাসের মতোই ব্যবহার করা যাবে অর্থাৎ, যেখানে মূল ক্লাস ব্যবহৃত হয়েছে, সেখানে সাবক্লাসও একইভাবে কাজ করবে। যেমনঃ  ধরি Bird নামে একটা ক্লাস আছে তার একটা মেথড হলো fly() আর এই ক্লাসের সাব ক্লাস হলো Magpie। আপনি চাইলে যেখানে Bird class ব্যবহার করবেন সেই জায়গাতে Bird এর পরিবর্তে Magpie ক্লাস ব্যবহার করতে পারেন। আবার ধরুন Penguin নামে একটা সাব ক্লাস আছে কিন্তু এইখানে fly() নামে কোন মেথড নেই তাহলে এখানে Liskov Substitution Principleএর নিয়ম ভঙ্গ হবে। সাবক্লাসগুলো এমন হতে হবে যাতে মূল ক্লাসের জায়গায় সেগুলো ব্যবহার করা যায়।   
  
I – Interface Segregation Principle :  বড় বড় ইন্টারফেসগুলোর বদলে ছোট ছোট ইন্টারফেস তৈরি করা উচিত যাতে সহজে প্রয়োজনীয় ফাংশনালিটি ব্যবহার করা যায়। একটি বড় ইন্টারফেসে অনেক ফাংশনালিটি থাকলে অন্যান্য ক্লাস গুলোতে সেই ফাংশানালিটি ইমপ্লিমেন্ট করতে হবে যা ওই ক্লাসের প্রয়োজেন নেই আবার এতে কোডের Complexity বেরে যায়। এজন্য ছোট ছোট  ইন্টারফেসে ভাগ করে কাজ করা । যেমনঃ কোন একটা রেস্টুরেন্টে অনলাইনে খাবার Order  করা যায় আবার তাদের অফলাইন আউটলেট আছে এখন সেই ক্ষেত্রে অনলাইন এবং অফলাইনের জন্য আলাদা আলাদা  ইন্টারফেস তৈরি করা কারন অনলাইনের অনেক ফাংশনালিটি আছে যা অফলাইনের ক্ষেত্রে প্রয়োজন পরবে না ।   
  
D – Dependency Inversion Principle :   High level module গুলো low level module  এর উপর সরাসরি নির্ভর করবে না। অর্থাৎ, নির্দিষ্ট কোনো ক্লাসের উপর নির্ভর না করে অ্যাবস্ট্রাকশন ব্যবহার করা উচিত। যেমনঃ ধরুন একটা send\_mail  ক্লাস আছে এখন তা যদি আরেকটা Gmail ক্লাসের উপর সরাসরি নির্ভর করে তাহলে এখানে Dependency Inversion Principle এর নিয়ম ভঙ্গ হবে কারণ মেইল পাঠানোর জন্য আমরা অন্যান্য সার্ভিস(Yahoo) ব্যবহার করতে পারি । এক্ষেত্রে আরেকটা Email নামে ক্লাস রাখতে পারি send\_mail ক্লাস ওই Email ক্লাসের ওপর নির্ভরশীল থাকবে আর Gmail, Yahoo যেই সার্ভিস ব্যবহার করতে চাই Email ক্লাসকে ইনহেরিট করে সেই নামে ক্লাস তৈরি করবে।