**SOLID PRINCIPLES**

**SOLID** is a mnemonic acronym for five principles: Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle.

**DESIGN PROBLEM:** Contain N x M square rooms, where N and M are the user-specified width and height of the maze**,** each room has at least one door to an adjoining room the program must be able to print the mazes using ASCII character or draw them in an image

**SINGLE RESPONSIBILITY PRINCIPLE:** This principle is very closely related to the more general principle of Cohesion, which says that the responsibilities of any component (method, class, sub-system, etc.) should be tightly aligned and focused on a single purpose

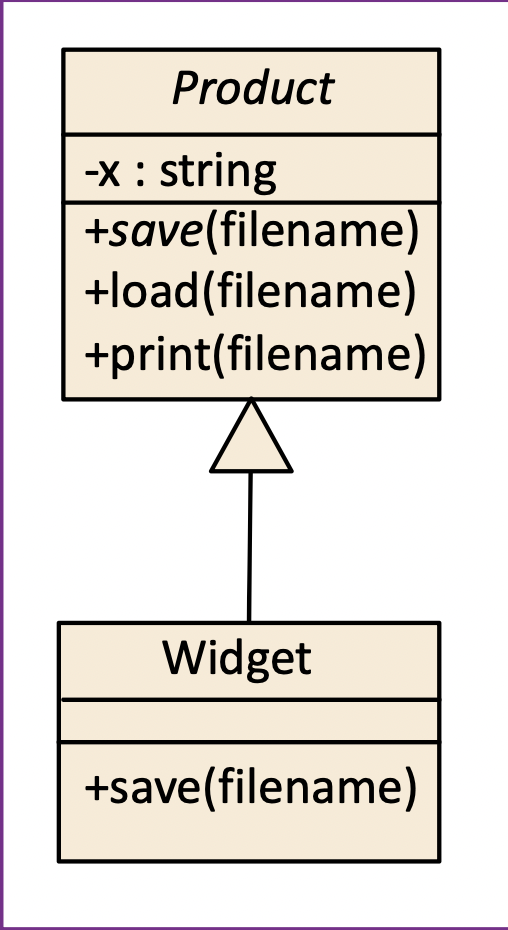
**OPEN/CLOSED PRINCIPLE:** A class is open if it is still available for extension a class is closed if it is available for use by other class, and therefore should not be modified

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES:** An abstract and a pure virtual class(C++)may include data members and some method implementations, the modern Open/Closed Principle encourages developers to use interfaces, abstract classes, and pure virtual classes to declare public data members

**OPEN/CLOSED PRINCIPLE:** Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes, Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically, this technique has been embodied in something called the strategy pattern – more on this later, use a generic to capture a template solution and instantiate it with the specific data types

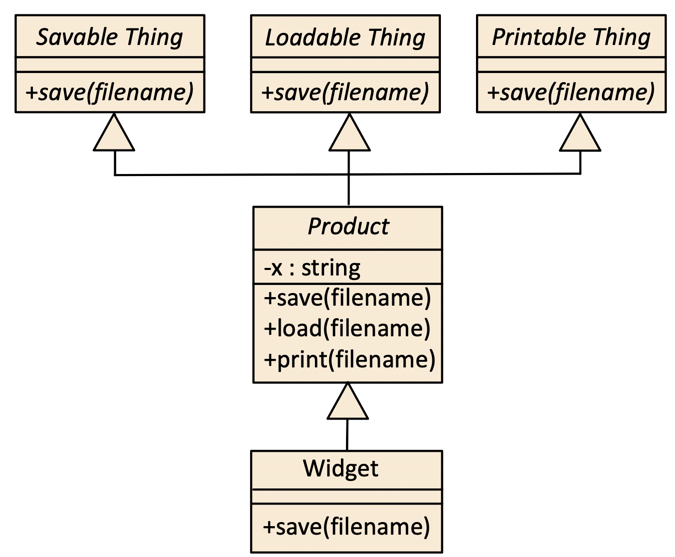
**LISKOV SUBSTITUTION PRINCIPLE:** if S is a specialization of T, then an S object can be used wherever a T object is required, Strong behavioral subtyping – an S object must be able to do everything any T object can do

Let Product be a base class, with one virtual method ,called save, whose intent is to save an object to a file widget, of some Product, ensure tha the implementation of save in Widget adheres to the purpose of save in Product



**INTERFACE SEGREGATION PRINCIPLE:** an interface represents public methods of a component, an interface doesn’t have to declare all the possible public methods of a

component: a component can have many interfaces, Following the Interface Segregation Principle, when used with other principles, can help Developers reduce complexity by increasing Cohesion and reducing Coupling

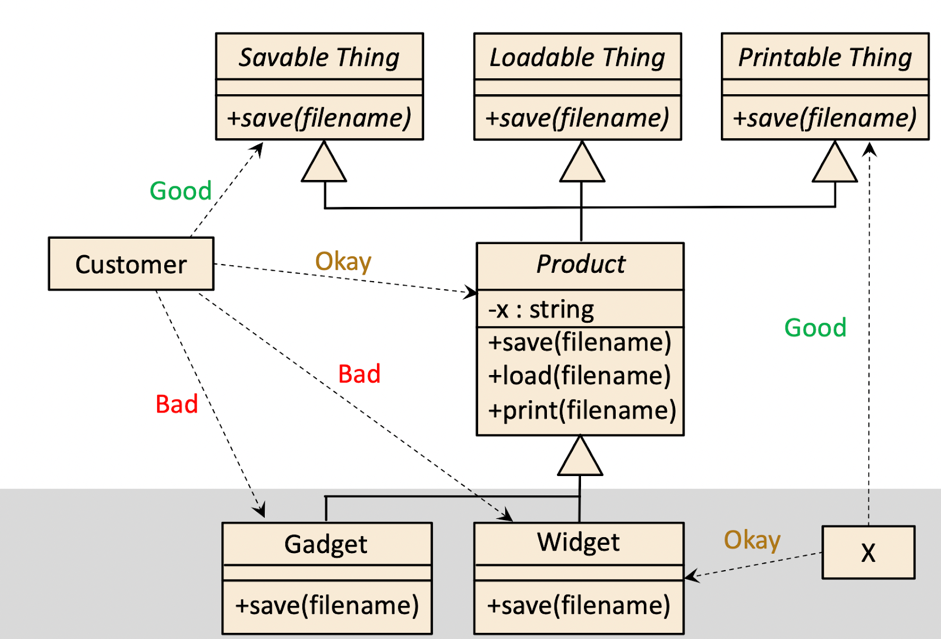


**DEPENDENCY INVERSION PRINCIPLE:** How to apply the Dependency Inversion Principle

• High-level modules should not depend on low-level modules

• Both low – level and high-level modules should depend on abstractions • “Program to the abstraction”

• Following the Dependency Inversion Principle helps Developers



**COMMON PARADIGMS**

Today, many languages and development environments support multi-paradigm software development, Object orientation (OO), Aspect orientation (AO), Functional programming (FP), Logic programming (LP), Genetic programming (GP) • Structured program (SP)

**PROBLEM BACKGROUND:** Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees, however, the concepts and definitions of these principles differ across paradigms.

**CONTRIBUTIONS OF THIS INITIAL PAPER:** Clarify the purpose of software-engineering (SE) principles, in general, and distinguish them from “best practices”, idioms, and patterns propose a template for documenting principles that allows a principle’s definition to go beyond just communicating the underlying concepts, provides a basis for assessing adherence to the principle and a foundation for teaching the principle to programmers

**SOFTWARE ENGINEERING PRINCIPLES:** a principle is a foundational concept (truth, proposition, rule, etc.) that leads to and supports reasoning about desirable characteristics, such as maintainability, efficiency, openness, reusability, etc.

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION:** Ensure that the private implementation details of a component are insulated so they cannot be accessed or modified by other components. Doing so will lead to better testability, maintainability, and reliability. It will also help with a clear separation of concerns and avoid accidental coupling.

**NON-REDUNDANCY AND COMPLIMENTARY – CRITERION:** Abstraction and encapsulation might be considered duals of each other, but one cannot subsume the other because the mechanisms for doing each are different, we show satisfaction of the second criteria, namely that developers and choose to follow each principle independent, with an example consisting of four functional-identical code snippets a simple program snippet with good Modularity, Abstraction, Encapsulation