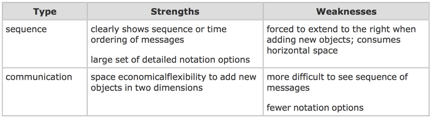
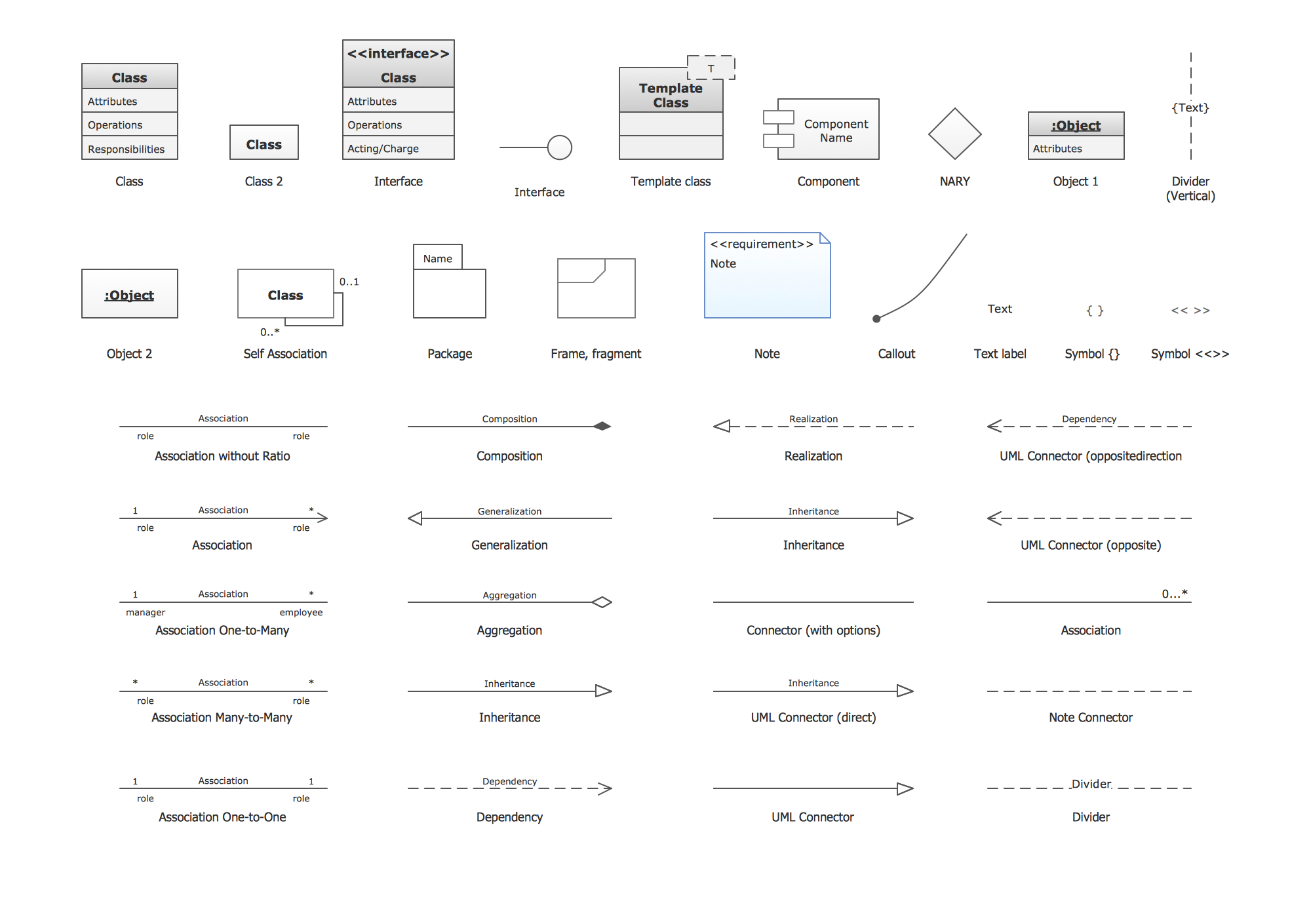
**VIII. Objected Oriented Design Principle**

1. **UML**

* **Concept**
* a general-purpose, developmental, modeling language
* intended to provide a standard way to visualize the design of a system
* UML Interaction Diagram



* **UML Class Diagram**

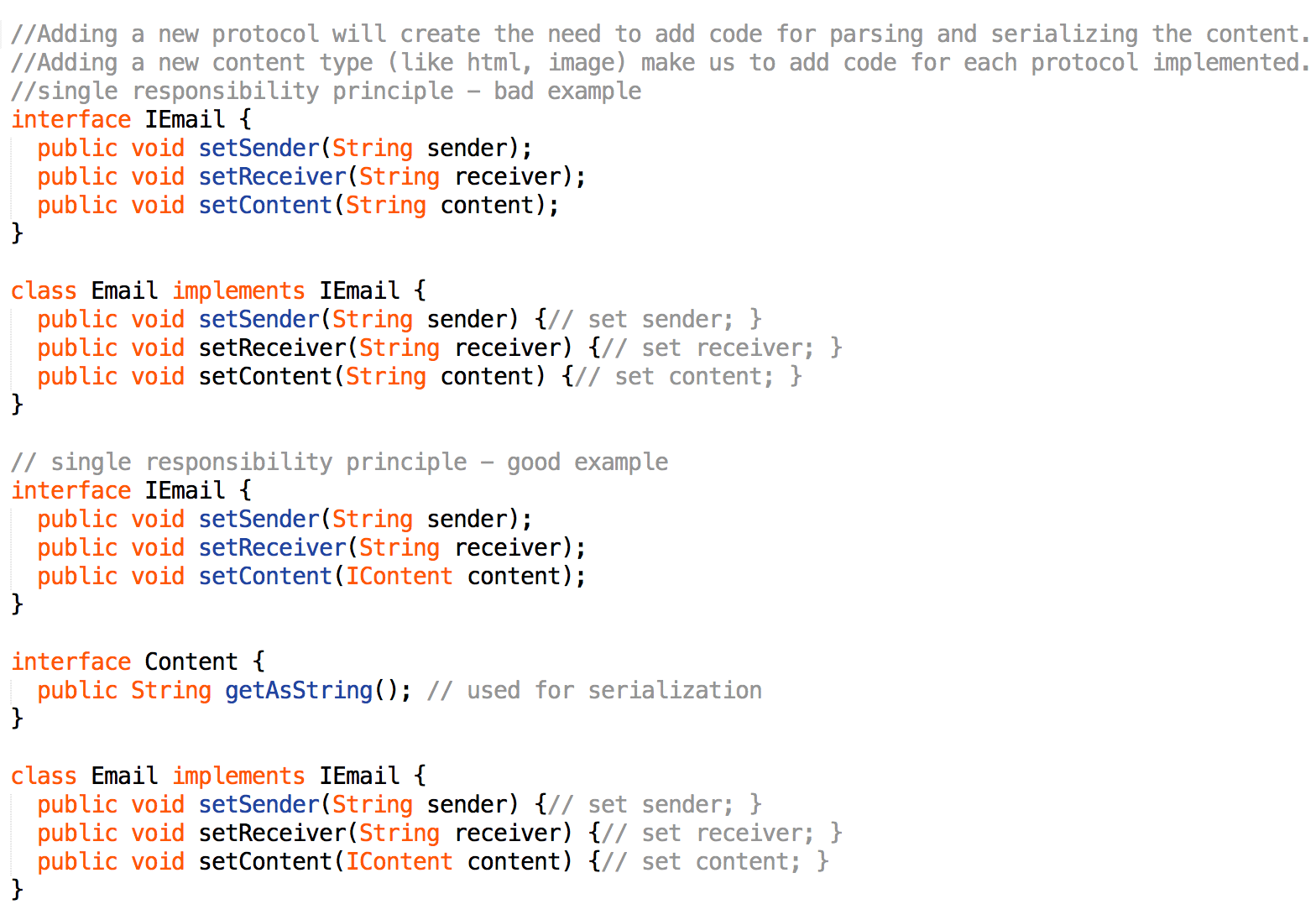
****

1. **SOLID**

* **Represent:**
* **Single responsibility**
* **open-closed**
* **Liskov substitution**
* **interface segregation**
* **dependency inversion**
* **Target**
* **Create a system easy to maintain and extend over time.**
* **Design Trap**
* **Rigidity**: simple modification lead to many module changes
* **Fragility**: simple modification lead to unrelated problem occurs
* **Immobility**: difficult to separate part of system
* **Viscosity**: hard code easier than maintain system design
* **Needless** **Complexity**: code contains unnecessary component
* **Needless** **Repetition**: copied similar coding lead to hard understand and maintain
* **Opacity**: unclear, not keep reconstructing code

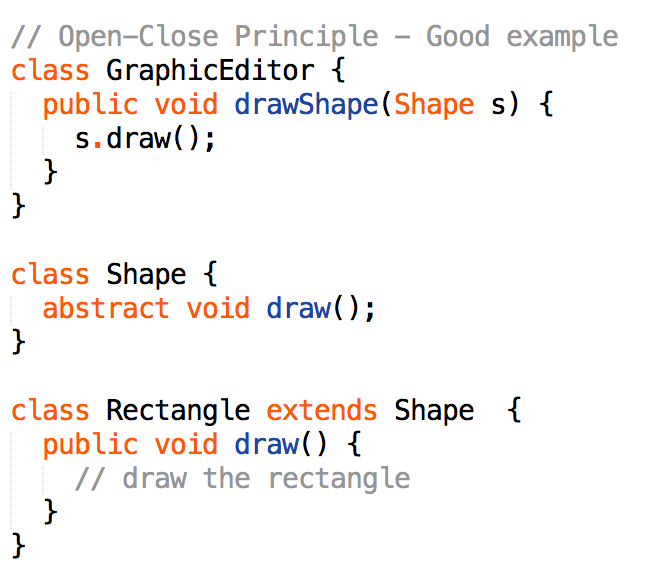
1. **SRP (Single responsibility)**

* **Principle**: for a class, only one reason leads to change
* **Why**: if a class take several responsibilities, all of them may lead to state change
* **Solution**: create another class to separate responsibility.
* **Note**: If one function change will lead to another function change, then unnecessary to separate
* **Example**:



1. **OCP (open-closed)**

* **Principle**: class, methods are open for extension, but closed for modification
* **Why**: if one modification leads to many module change, it causes rigidity.
* **Solution**: system reconstruct with abstraction (use interface or abstract class)
* **Note**:
* Only use abstract if the module changes frequently
* **Bad** design: add Triangle will lead to DrawAllShape() method change by add more if/else
* **Good** design: Use interface, extended class Triangle override DrawAllShape()

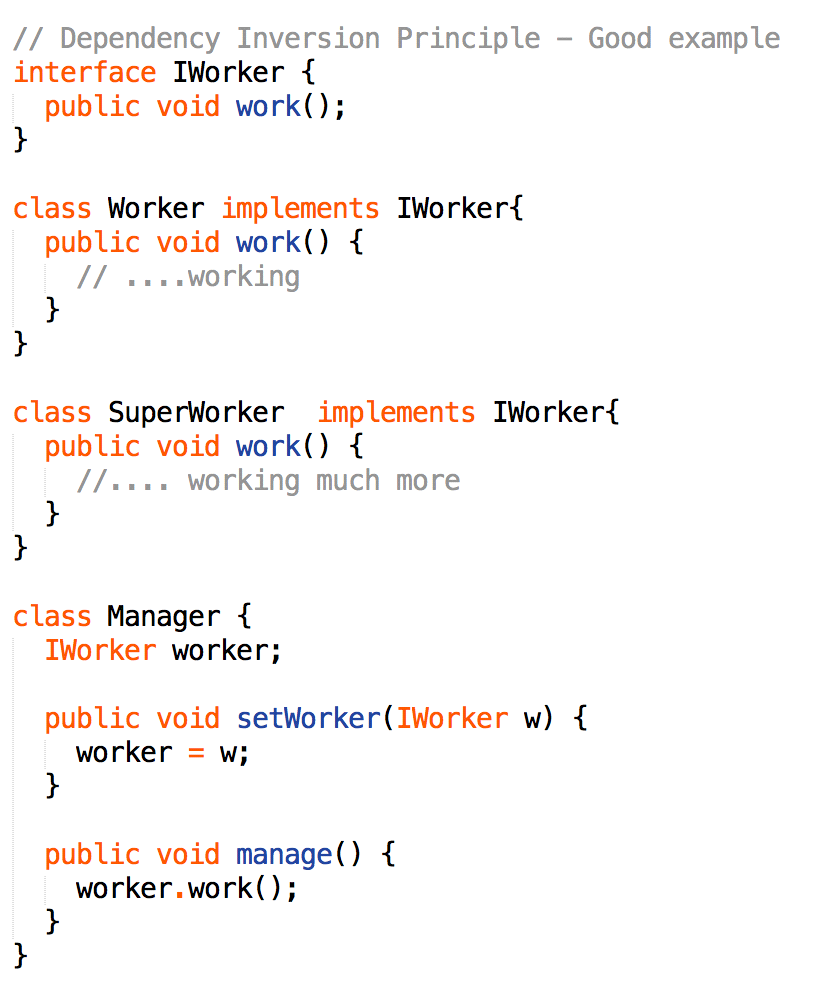
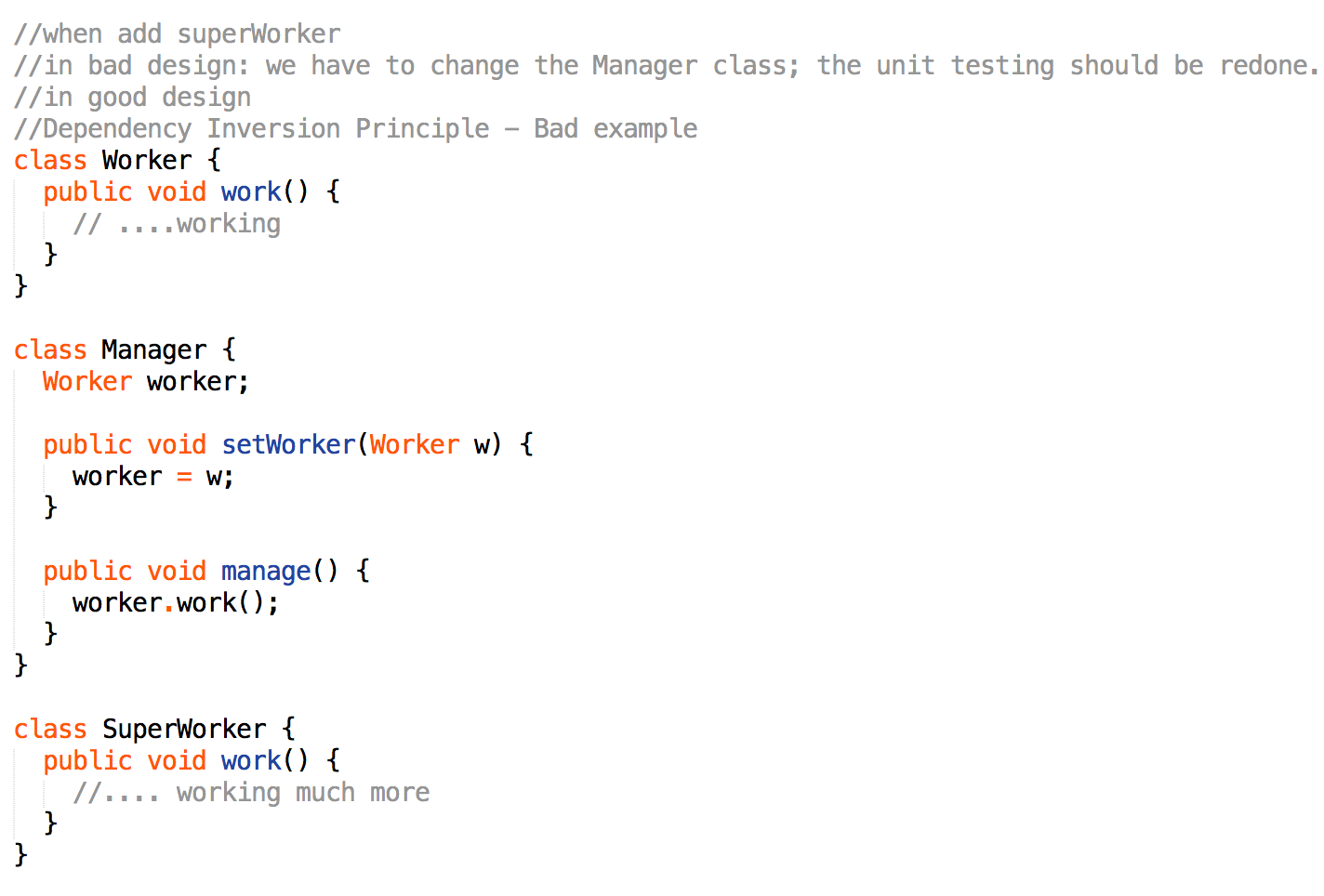


1. **LSP (Liskov substitution)**

* **Principle**: subtype must be able to substitute its parent class (we must make sure that new derived classes are extending the base classes without changing their behavior)
* **Why**: keep code maintainable, reusable, robust
* **Solution**: do not use inheritance if no behavior IS-A relationship between (like: square should not extends rectangle)
* **Note**: Best solution is to create a layer structure make square and rectangle siblings

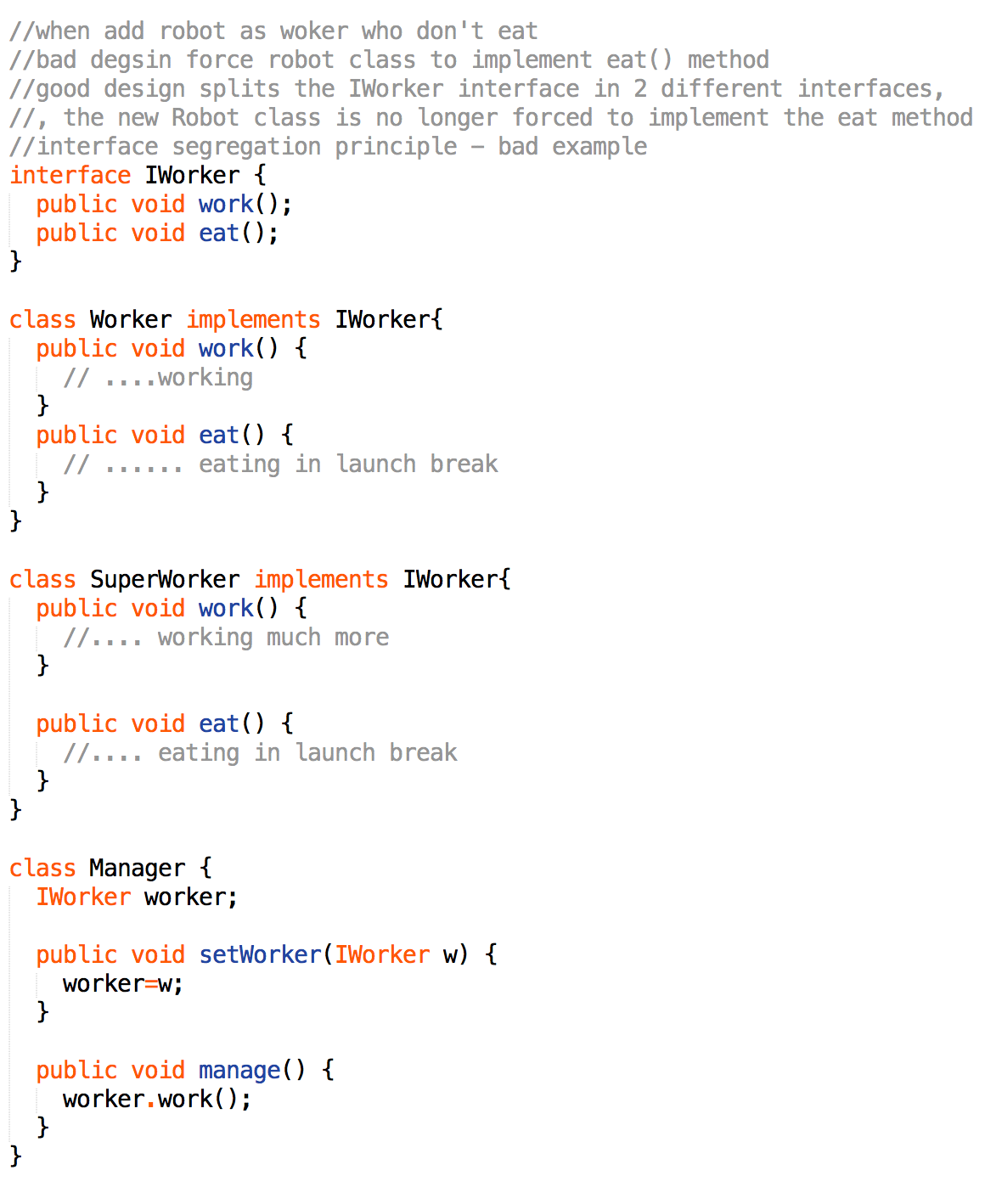
1. **DIP (dependency inversion)**

* **Principle**:
* higher-level module should not depend on lower-level module
* both should depend on abstraction
* abstract should not depend on detail
* detail should depend on abstraction
* **Why**: a bad design the high-level class uses directly and depends heavily on the low-level classes
* **Solution**: introduce an abstraction layer between high level classes and low-level classes
* **Example**



1. **ISP (Interface segregation)**

* **Principle**: Clients should not be forced to depend upon interfaces that they don't use.
* **Why**: it will cause fat interface or polluted interface
* **Solution**: Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one sub-module.
* **Example**

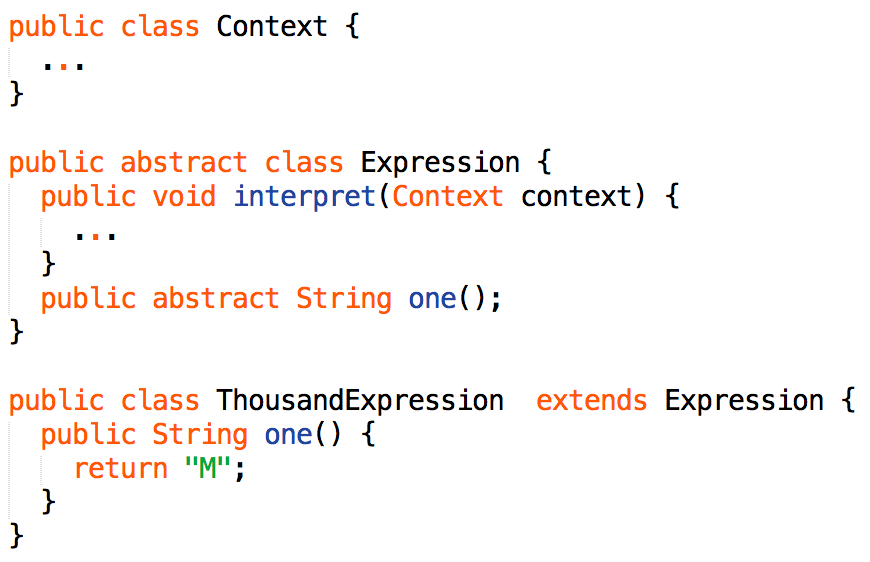


1. **Iterator Pattern**

* **Intent**: Provide a way to access the elements of an aggregate object sequentially without exposing its internal structure.

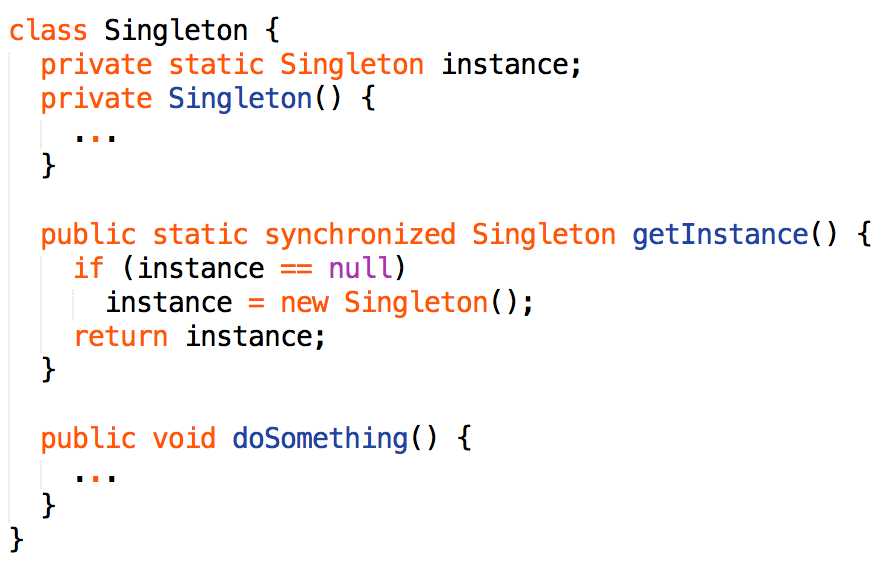
1. **Interpreter Pattern**

* **Intent**: Given a language, define a representation for its grammar along with an interpreter



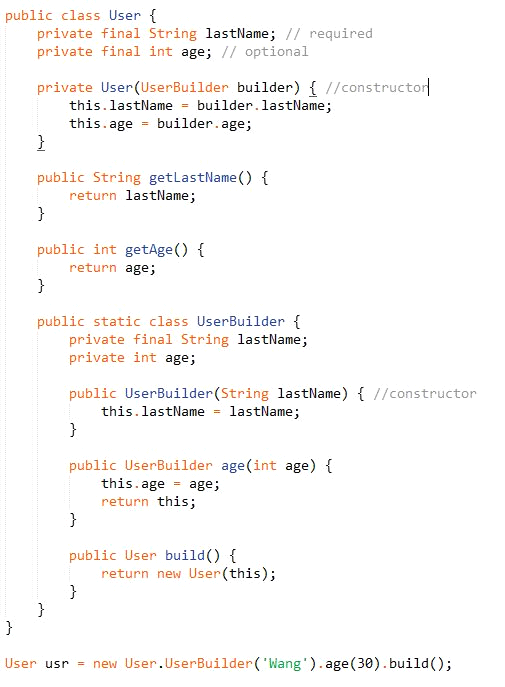
1. **Singleton Pattern**

* **Intent**:
* Ensure that only one instance of a class is created.
* Provide a global point of access to the object.
* **Solution**: The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member
* **Example**



1. **Builder Pattern**

* **Intent**: When you want to create an object that has many fields, using constructors can become unwieldy and confusing. Escpeically when some attributes are required and some optional.

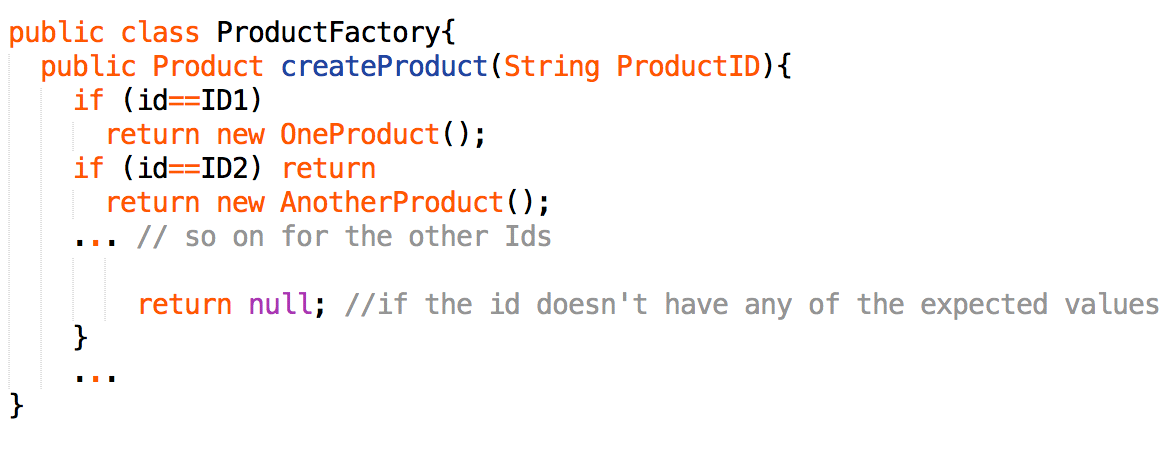


1. **Adapter Pattern**

* Intent:
* Convert the interface of a class into another interface clients expect.
* Adapter lets classes work together, that could not otherwise because of incompatible interfaces.

1. **Factory Pattern**

* **Intent**:
* creates objects without exposing the instantiation logic to the client.
* refers to the newly created object through a common interface
* **Example**



1. **Factory Method**

* **Intent**:
* Defines an interface for creating objects, but let subclasses to decide which class to instantiate
* Refers to the newly created object through a common interface
* **Example**:

