Components and Concerns

In the context of this experiment, software design is the overall organization of functionalities into methods, classes, relationships and components (or packages). A **component** is a logical structure that groups classes related to a common concern. In other words, a component represents a single concern (feature), which in turn is implemented by a group of classes.

A **concern** is often the conceptual representation of a feature implemented in a component. However, a concern may also be scattered in several components, in which it is not the main concern. We call this kind of concern by cross-cutting concerns, since it cross-cut the implementation of other concerns. A common example of cross-cutting concern is Exception Handling. This handling of exceptions is often scattered in different components.

Object-oriented Software Design Principles

Software design principles are principles that help in the design of modules aiming at best modularize the implementation of a system. The violation of such principles often increases the maintainability cost of the system. Table I a brief description of each software design principle.

TABLE I. List of well-know Design principles

Name	**Description**
Open- Closed	A class should be extensible without need to change it
The Single Responsibility Principle	Each class should have only one reason to change
The Liskov Substitution Principle	Derived classes must be substitutable for their base classes
The Interface Segregation Principle	Each interface should target a specific type of client components
The Dependency Inversion Principle	Depend on abstractions, not concretions

Design Problems description

A design problem (or a design smell) represents the realization of either: (i) unintended design decisions, which violate the original, intended design of a system, or (ii) violations of well-known software design principles. Unwanted dependency is a type of design problem falling in the first category because it introduces an inter-component dependency that is not part of the intended design. **Fat interface** is an example of a design problem in the second category as it violates design principles, such as Interface Segregation and Single Responsibility principles. These both types of design problems are high-level structures that often affect multiple elements in the source code. TABLE II presents a list of well-known design problems. This experiment is not limited to detecting only these types. However, we encourage you to use the list of design problems as a reference guide. You may detect other design problems that are not part of the list below.

TABLE II. List of well-known Design Problems

Type	**Description**
Ambiguous Interface	Interfaces that offer only a single, general entry-point into a component.
Fat Interface	Interface of a design component that offers only a general, ambiguous entry-point that provides non-cohesive services, thereby complicating the clients' logic.
Component Overload	Design components that fulfill too many responsibilities.
Cyclic Dependency	Two or more design components that directly or indirectly depend on each other.
Scattered Concern	Multiple components that are responsible for realizing a crosscutting concern.
Delegating Abstraction	An abstraction that exists only for passing messages from one abstraction to another.
Overused Interface	Interface that is overloaded with many clients accessing it. That is, an interface with "too many clients".
Unused Abstraction	Design abstraction that is either unreachable or never used in the system.
Unwanted Dependency	Dependency that violates an intended design rule.

Code Smells Definitions

Code smells are symptoms in the source code that may indicate maintainability problems, such as design problems. Code smells are not bugs, instead they only indicate weakness in the source code design that may cause maintainability problems or increase the risk of bugs and failures in the future. Several types of code smells were investigated and catalogued by researchers and practitioners. Table III shows a short description for each type of code smell considered in this experiment.

Table III. Code Smell Definitions

Type	**Description**
God Class	Long and complex class that centralized the intelligence of the system

Brain Method	Long and complex method that centralizes the intelligence of a class
Data Class	Class that contains data but not behavior related to the data
Disperse Coupling	The case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes
Feature Envy	Method that calls more methods of a single external class than the internal methods of its own inner class
Intensive Coupling	When a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes
Refused Parent Bequest	Subclass that does not use the protected methods of its superclass
Shotgun Surgery	This smell is evident when you must change lots of pieces of code in different places simply to add a new extended piece of behavior
Tradition Breaker	Subclass that provides a large set of services that are unrelated to services provided by the superclass

Categories of Agglomeration

A code smell agglomeration is a group of inter-related code smells that may indicate the full extension of a design problem (Oizumi, 2016). The relations among code smells are determined by the relationships that exists between the smelly code elements. In our context, a smelly code element is an element that is affected by a code smell. Relations among smelly code may assume different forms, according to the agglomeration category. Below, on Table IV, we present a brief definition of each category

Table IV. Agglomeration Categories Definitions

Category	**Description**
Intra-component	A component that contains two or more classes affected by the same type of smell i.e. occurrences of the same type of code smell are located within classes

	of a single component
Hierarchical	Two or more classes in a common inheritance tree (including interface implementation) that are affected by the same type of smell
Concern Overload	Classes that implement one or more crosscutting concerns besides implementing