Components, Connectors 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.

Two different components communicate with each other using **connectors**. A connector is a design element that models interactions among components and the rules that governs those interactions. Examples of connectors are procedure calls, shared variables access, client-server protocols and asynchronous event multicast.

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 increase the maintainability cost of the system.**Ошибка! Источник ссылки не найден.** shows a brief description of each software design principle. If you need more information about software design principles, please consult the Principles and Patterns document.

Table I. Principles of OO Software Design

Name	Description
The Single Responsibility Principle	A class should have one, and only one, reason to change
The Open Closed Principle	You should be able to extend a classes behaviour, without modifying it
The Liskov Substitution Principle	Derived classes must be substitutable for their base classes
The Interface Segregation Principle	Make fine grained interfaces that are client specific
The Dependency Inversion Principle	Depend on abstractions, not concretions

Software Design Problem

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 dependencyis 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 IIpresents 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
Unwanted Dependency	Dependency that violates an intendeddesign rule.
Fat Interface	Interface of a design component that offers only a single, general entry-

	point, but provides two or more functionalities.
Concern Mixing	Component that mix aconnector-related concern with other concerns of the system.
Cyclic Dependency	Two or more design components that directly or indirectly depend on each other.
Multiple Interaction Types	Two different connectors that are used to link the same pair of components.
Scattered Concern	Multiple components that are responsible for realizing the same design concern.
Overused Interface	Interface that is overloaded with many clients accessing it. That is, an interface with too many clients.
Unused Interface	Interface that is never used by external components.

Code Anomaly

Code anomalies are symptoms in the source code that may indicate maintainability problems, such as design problems.Code anomalies 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 anomalies were investigated and catalogued by researchers and practitioners. However, in this experiment we focused in a set of 10 types, which were catalogued by Lanza and Marinescu (2006). Table III shows a short description for each type of code anomaly considered in this experiment.

Table III. Types of Code Anomaly

Type	Description
Brain Class/God Class	Long and complex class that centralizes 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 inthe system, and additionally these provider methods that are dispersed amongmany classes
Feature Envy	Method that calls more methods of a single external class thanthe internal methods of its own inner class
Intensive Coupling	When a method is tied to many other operations in the system, whereby theseprovider 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 anomaly is evident when you must change lots of pieces of code in differentplaces simply to add a new or extended piece of behavior
Tradition Breaker	Subclass that does not specialize the superclass