Refactoring for Software Design Smells

Abstraction

Missing Abstraction

This smell arises when clumps of data or encoded strings are used instead of creating a class or an interface.

Imperative Abstraction

This smell arises when an operation is turned into a class.

Incomplete Abstraction

This smell arises when an abstraction does not support complementary or interrelated methods completely.

Multifaceted Abstraction

This smell arises when an abstraction has more than one responsibility assigned to it.

Unnecessary Abstraction

This smell occurs when an abstraction which is actually not needed (and thus could have been avoided) gets introduced in a software design.

Unutilized Abstraction

This smell arises when an abstraction is left unused (either not directly used or not reachable).

Duplicate Abstraction

This smell arises when two or more abstractions have the identical name or identical implementation or both.

Modularization

Broken Modularization

This smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions.

Insufficient Modularization

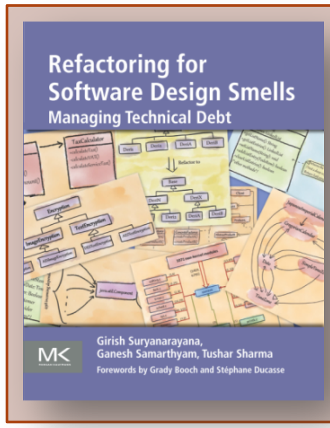
This smell arises when an abstraction exists that has not been completely decomposed and a further decomposition could reduce its size, implementation complexity, or both.

Cyclically- dependent Modularization

This smell arises when two or more abstractions depend on each other directly or indirectly (creating a tight coupling between the abstractions).

Hub-like Modularization

This smell arises when an abstraction has dependencies (both incoming and outgoing) with large number of other abstractions.



"Refactoring for Software Design Smells: Managing Technical Debt", Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma, ISBN: 978-0128013977, Morgan Kaufmann/Elsevier, 2014

http://amzn.com/0128013974

Encapsulation

Deficient Encapsulation

This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required.

Missing Encapsulation

This smell occurs when the encapsulation of implementation variations in a type or hierarchy is missing.

Unexploited Encapsulation

This smell arises when client code uses explicit type checks(using chained if-else or switch statements) instead of exploiting the variation in types already encapsulated within a hierarchy.

Leaky Encapsulation

This smell arises when an abstraction "exposes" or "leaks" implementation details through its public interface.

Hierarchy

Missing Hierarchy

This smell arises when a code segment uses conditional logic (typically in conjunction with "tagged types") to explicitly manage variation in behavior where a hierarchy could have been created and used to encapsulate those variations.

Unnecessary Hierarchy

This smell arises when the whole inheritance hierarchy is unnecessary, indicating that inheritance has been applied needlessly for the particular design context.

Unfactored Hierarchy

This smell arises when there is unnecessary duplication among types in the hierarchy.

Wide Hierarchy

This smell arises when an inheritance hierarchy is "too" wide indicating that intermediate abstractions may be missing.

Speculative Hierarchy

This smell arises when one or more types in a hierarchy are provided speculatively (i.e. based on imagined needs rather than real requirements).

Deep Hierarchy

This smell arises when an inheritance hierarchy is "excessively" deep.

Rebellious Hierarchy

This smell arises when a subtype rejects the methods provided by its supertype(s).

Broken Hierarchy

This smell arises when a supertype and its subtype conceptually do not share an "IS-A" relationship resulting in broken substitutability.

Multipath Hierarchy

This smell arises when a subtype inherits both directly as well as indirectly from a supertype leading to unnecessary inheritance paths in the hierarchy.

Cyclic Hierarchy

This smell arises when a supertype in a hierarchy depends on any of its subtypes.

http://www.designsmells.com/