Code Smells (Indicators of Poor Design)

Code Smell	Reasoning	Solution
Duplicated Code This is fairly obvious – identical code which is repeated, often through copy and paste.	Duplicated code increases the size of the overall code base. Changes to one copy need to be applied to all the other copies – this is a maintenance burden.	Create a function which captures the behaviour of the duplicate code. Ensure that this function is called whenever this behaviour is needed. Follow the DRY principle.
Long Function Long functions stick out because they are substantially longer than other functions in the code base. Of course, if *all* your functions are long you have more serious problems.	Long functions are difficult to understand, test and debug.	Split the function up using smaller, possibly private helper functions. Consider whether some of the behaviour encapsulated in the function actually belongs in other classes.
Monolithic or Large Class Classes which have an extraordinary number of functions and data members, or a number of very long functions (see Long Function). Classes with diverse responsibilities.	A monolithic class is likely to embody several distinct concepts. This makes the class difficult to understand and dilutes encapsulation. Monolithic classes are harder to make correct and error-safe because they tackle multiple responsibilities.	Divide and conquer – identify the different concepts captured within the monolithic class and pull related behaviours into their own classes.
Data Class Data classes offer lots of getter and setter functions but no real behaviour. Their behaviour is out there somewhere, scattered among the rest of the code.	The presence of data classes often indicates a procedural approach to the design. Classes are merely used to package data together and do not offer fully-fledged abstractions. This forces clients to define the behaviour (see Duplicated Code). Clients are tightly coupled to the implementation details as there is little encapsulation.	Identify sites where clients are providing behaviour that actually belongs in the data class. Move these behaviours into the class.

Inappropriate Intimacy Classes are able to directly access and *modify* each others' private parts, often through getter/setter functions.	Private data is the best means that a class can use to preserve its invariants. However, if this data is exposed through the class's interface there is no guarantee that the invariants can be preserved. Additionally, clients become coupled to the internal representation.	Where possible remove implementation-revealing parts of a class's interface. Encapsulate appropriate behaviour within the class rather than providing getter functions, and follow the "Tell, don't ask" principle.
Switching Based On Type Codes Classes (usually within an inheritance hierarchy) store a specific code indicating what type they are. Switch statements or control logic is used to determine the course of action based on the type code.	In object-oriented programming, the type of an object is represented by the way it behaves, not by its state. Having such switch statements in the code base introduces a maintenance burden. Every time a new class is added to the hierarchy the switch statements or control logic will need to be updated to handle the new type.	Polymorphism provides an elegant, object-oriented solution to this problem. Implement type-based decisions by using virtual functions and dynamic binding, not with conditional control structures.
Refused Bequest Derived classes inherit the member functions and data members of their parents. Some of these classes do not want what they have been given. This is especially problematic if the derived class is refusing to fulfil the contract of the base class.	Inheritance should model an is-substitutable-for relationship. Having an interface that is not a true reflection of what a derived class does, or does not make sense for a derived class, results in confusion and makes the code harder to use and understand.	Re-think the inheritance hierarchy - push inappropriate member functions and data members out of the base class and into derived classes. Alternatively, replace inheritance with composition – a containment relationship.
Public Inheritance Solely For Code Reuse No code is written to the interface of the base class of an inheritance hierarchy. There are no virtual functions in the base class or the base class has virtual functions which are never overridden.	Using inheritance so that a derived class can reuse base class code to implement itself results in brittle, unnatural and inefficient designs. Brittle, because inheritance is a highly coupled relationship and so it becomes difficult to change the base class implementation without breaking derived classes. Unnatural, because implementation details in the base class are often not common to all derived classes; and inefficient, because clients end up having to include excess header files.	Implemented-in-terms-of relationships can be entirely proper but should be modelled using composition, or, possibly, private inheritance. Use public inheritance to model roles in the system and the objects substitutable for these roles (see the Liskov Substitution Principle).