Andrei Fangli
Class 258
R3: research paper
26th of December 2015

Design by Contract

This paper bridges a number of principles that are critical in OBJECT ORIENTED DESIGN and illustrates how DESIGN BY CONTRACT can help the developer follow them. The goal is to gain a more in depth understanding of OBJECT ORIENTED PROGRAMMING, knowing how inheritance relates to this programming paradigm and how it can differ from other domains.

While DESIGN BY CONTRACT can help a great deal its lack of support in external system such as SQL and other Web APIs can prove difficult if not impossible to have a software that fully uses the concept.

Last but not least, a great advantage would be the explicit use of a validation service to validate domain entities and report errors back to the user about what is wrong about them.

1 Designing by Contract

CONTRACTS in software refer to specifying constraints for objects that either must be hold when calling a method or they ensure that the result of a method fulfils a number of criteria. PRECONDITIONS are constraints for objects that are provided to a method to be used for an operation. POSTCONDITIONS are constraints that are satisfied by the result or results of a method. INVARIANTS are constraints that are satisfied both when calling the method and when the result is returned. [1]

In OBJECT ORIENTED PROGRAMMING, inheritance is one way of reusing code. Either by deriving and overriding methods to extend what they do or by defining method on base class types and use derived type instances interchangeably. Both methods are related to SOLID[1] principles.

The OPEN-CLOSED PRINCIPLE can be summarized to *"Software entities should be open for extension, but closed for modification"* [2]. This way all defined classes that are tested and considered complete may not be changed in order to add or change functionality for an application thus eliminating maintenance and retesting for them. Any change to the functionality of an application is done either by extending existing classes or defining new ones that may later on be extended and thus maximizing code reuse. [2]

This is the most difficult principle from SOLID to go by because class inheritance can be difficult to understand and use appropriately. Initially when OBJECT ORIENTED PROGRAMMING was starting to become more and more popular it was though that inheritance is the idea way of reusing code. As years went by and experience was gained by dealing with problems that came from overuse of class inheritance it became clear that this technique is not the easiest or safest to reuse code. [3]

A principle that stands aside from SOLID, but should always be followed states *"Favor composition over inheritance"* [3]. Through experience, developers have found out that in order to maximize reuse in a safe and easy meaner is to first attempt to define new objects in terms of other existing ones through composition rather than inheritance. This may lead to a different problem where

[1] SOLID – This is an acronym introduced by Michael Feathers for five core principles for Object Oriented Design. The first letter of each principle is used to form the acronym: Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle and Dependency Inversion Principle.

a lot of objects are used to create a new one thus gravely limiting the reuse potential. As any client that would try to reuse the new object would have to know about all object that it depends upon. This known as HIGH COUPLING or composition overuse. [3]

Even if **HIGH COUPLING** may be better than creating broken class hierarchies it is essential to ask why is inheritance difficult to understand? It is obvious that something is missing from the principles that are in books about **OBJECT ORIENTED PROGRAMMING** in order to help make class inheritance more effective. Both composition and inheritance are tools, in order to know when to use rather than the other one must understand them both with their advantages and disadvantages.

1.1 Inheritance Controversy

In general class inheritance is summarized to the IS A relationship. If one can write that *something* IS A *something else* then between the two can be modeled using inheritance. One can say that a QUEUE is a LIST as the former restricts how elements are added and removed to the latter. Everything works until one has to implement a QUEUE by inheriting a LIST and quickly realizes that an insert method makes no sense to a queue, the order in which elements are added is very specific in contrast to its presumed base class. One can remove a given element from a LIST while one removes an element from a QUEUE by extracting the earliest existing added item. Slightly different, suddenly something has gone terribly wrong through use of class inheritance.

What characterizes an object? They all have an IDENTITY, something that makes them distinguishable regardless of their STATE which is *encapsulated* and should only be accessible through *methods* that resemble BEHAVIOR. What makes an object important or useful is what it does, its BEHAVIOR. A QUEUE may be implemented using a LIST however it does not extend it. From a behavior point of view, a QUEUE is not a LIST because a QUEUE does not add behavior to a LIST, it limits it. Nor does does it change how a LIST does what it does.

For inheritance to work it must not break its base class, how can this be done with a handful of method signatures? It is often said that *"Inheritance breaks encapsulation"* [3]. When one overrides a method, he or she is mostly looking at the validation of arguments that is described at the beginning of the method in the majority of cases. This is done in order to know what constraints the arguments satisfy after the base implementation has been called or to know what is ensured about the result of the base implementation. In both cases the interest is what can be provided to the base implementation in order to know the maximum level on constraints set on input and to know the minimum level of constraints set on the output of the method. These both are done to ensure that any SUBCLASS instance can replace an instance of the BASECLASS without breaking any existing functionality. The encapsulation is broken only within the hierarchy and not in all methods that make use of those instances. [4]

This introduces yet another principle from SOLID. The LISKOV SUBSTITUTION PRINCIPLE which states that an object may substitute other objects of its base type without altering any properties of a program. When a program, or in more modern terminology, an application is defined in terms of a base type, then only what instances of that type know what to do with input, with their constraints, and what output they ensure is known. Any instance of a subtype must do the same if it is to substitute instances of its base type. [4]

Again the focus is on what constraints input and output objects must satisfy in order to have a correct subtype. This boils down to PRECONDITIONS, POSTCONDITIOSN and INVARIANTS. If they were to be explicit and part of the method signature it would reduce the need to break encapsulation when defining derived classes. Following the LISKOV SUBSTITUTION PRINCIPLE the subtype instances could substitute their base type instances because all constraints on output objects must be fulfilled.

Having the exact same PRECONDITIONS, POSTCONDITIONS and INVARIANTS throughout a hierarchy makes it very rigid, less flexible and thus less reusable. Does the subtype really need all the PRECONDITIONS? What if it needs only a part of them? Would it allow instances to substitute their base type instances? Having less constraints may widen the range of possible input objects while still allowing those that base type instances would allow. In other words, it will not break as any value that worked for the base type instance will work for the subtype instance. [4]

What if PRECONDITIONS become more restrictive? They may reduce the range of possible input objects thus some of them that worked for a base type instance will not work for a subtype instance thus in some cases a property of the application may become altered. [4]

Can POSTCONDITIONS be less restrictive? They cannot, widening the range of possible output objects may allow the subtype to return an object that is not covered by one its clients and thus it may behave unexpected. [4]

Can POSTCONDITIONS be more restrictive? Reducing the range of possible output objects will still return objects that fulfil constraints specified by the base type. It will not break. [4]

INVARIANTS are constraints that are satisfied both when the method is called and when the method returns. They may disappear while deriving but their constraints must remain, for any INVARIANT that is removed there must be at least one PRECONDITION that is at most as restrictive as the INVARIANT and at least one POSTCONDITION that is at least as restrictive as the INVARIANT. INVARIANTS are a set of PRECONDITIONS and POSTCONDITIONS on the same objects thus INVARIANTS are a special case of the above mentioned. [4]

1.2 A Domain Specific Language

DOMAIN SPECIFIC LANGUAGES are languages created to solve a class of problems in contrast to GENERAL PURPOSE LANGUAGES that aim to solve multiple classes of problems. Examples of the former are CASCADING STYLE SHEETS, REGULAR EXPRESSIONS and so on, examples of the latter are the most common programming languages C++, C#, JAVA as well as UNIFIED MODELING LANGUAGE which is a language to modeling software systems (and not only). [5]

Such a language for specifying contracts would be appropriate as the aim here is to describe constraints which can be viewed as functions that need to return BOOLEAN results. Given a set of input parameters one must tell whether they are valid or not, eventually providing an error message that describes how input should be in order to use the method. Reiterating over what is required, given a number of input objects the GOAL is to determine whether a method can be called or not. This is a special case of RULE-BASED PROGRAMMING which relates to LOGIC PROGRAMMING. RULES or PREDICATES that declaratively describe what constraints input and output objects must fulfil.

Knowing that the main focus when defining contracts is actually writing predicates then the aim of the DOMAIN SPECIFIC LANGUAGE is to help describe them in an easy to understand fashion. Prolog could be used for this task however its lack of standard and various implementations will make it difficult to relate to. The result would be a language that is Prolog but not quite.

Another great issue that PROLOG has is its lack of type safety. While dynamic languages such as JAVASCRIPT take advantage of this it is not long until the developers writing their program start using naming conventions such as the HUNGARIAN NOTATION so they can keep track of the types of their variables. Dynamic languages are great for small projects, but for ENTERPRISE LEVEL applications they can significantly slow the development with every new set of classes.

PROLOG covers a lot of ground for LOGIC PROGRAMMING. For contracts, having variable terns, or input-output terms, does not make much sense. Methods assume the PRECONDITIONS as facts as well as the POSTCONDITIONS of other methods. For contracts, there is no need to use all that PROLOG has to offer. Having complex constraints will make them hard to maintain and extend. Besides most constraints will check whether a number is in range, whether an object is not null or a more complex scenario checking whether a collection contains null values.

DOMAIN DRIVEN DESIGN has introduced a number of concepts that bring the developers and domain experts more close. They must communicate with one another to establish how the software systems will help them automate their work. [6]

The UBIQUITOUS LANGUAGE is meant to be a bridge of communication between all stakeholders of a project. Each project has its own UBIQUITOUS LANGUAGE as the application domain can be vastly different or the area of the domain varies from application to application. This language should use domain terms from the domain experts as well as software terms such as page, cache and so on. The purpose of the language is to help everyone involved understand the solution, not in all its technical depth, but the high level idea behind it and most important, the domain experts must be able to tell whether the solution makes sense for their business. [6]

One important aspect of DOMAIN DRIVEN DESIGN is the concreteness of the UBIQUITOUS LANGUAGE. Each term is well defined, understood and can be looked up. For instance, for a software for accountants the UBIQUITOUS LANGUAGE would contain terms such as *period end*, *variance*, *reconciliation*, *accounts* and so on. For software developers each term must map to a class definition that define what properties, with their type, it has. This is the point where communication starts, not everything an *account* may have is useful to the software therefore the concept is abstracted into the software. Only what is necessary and makes sense to the application is kept. [6]

It is difficult if not impossible for software developers to understand the business domain in its whole depth, the developers need to come up with a rigorous and simple way to present their understanding of the domain and how the software models it to the user requirements. One way to do this would be to use UML[2] diagrams. The CLASS DIAGRAM can be used to illustrate the DOMAIN ENTITIES, DOMAIN VALUES and DOMAIN SERVICES to the domain experts, they are easy to understand and follow. If

[2] UML – Unified Modeling Language, it is mostly a graphical language that helps illustrate what a software system does and how it is structured, interacts and how it is deployed.

the domain experts do not see their business as the diagram shows it, then there must be a misunderstanding somewhere. [6]

The CLASS DIAGRAM can hide an important aspect, constraints. They are a critical part of the application as not just any value will suffice. Usually constraints are expressed as textual and enough times they lack depth, do not cover all cases or are interpretable. The luxury of a CLASS DIAGRAM is that it clearly illustrates the definition of a class. Depending on the implementing programming language there may be a number of GET/SET methods for accessing the attributes (JAVA, C++) or may have properties that have at least one of the two accessor methods (COMMON LANGUAGE SPECIFICATION compliant languages such as C#, VISUAL BASIC .NET, F# and so on). This is a minor technical detail that does not change how the software models the business in any way.

One way to eliminate the ambiguity of constraint specification is to have a formal language. One that is clear and leaves no room for interpretation. Besides the technical null checks, the business has constraints of its own that must be modeled as well. [6]

For instance, when an *accountant* wants to check the *variance* between two *accounting periods* they must be both exist in the system. In addition to that obvious constraint the *variance* can be carried out only on the same *account* because comparing two *reconciliations* done on different accounts does not have BUSINESS VALUE. The two selected *accounting periods* must have *reconciliations* for that *account* in an accepted state therefore not any *accounting periods* will do. All these constraints seem obvious for a business expert, however for a developer it is mostly data and not information.

Having a robust language for specifying domain constraints for models can allow it to be translated into a programming language, if not use the programming language itself and have the constraints generated rather than manually translated.

Coding constraints is beneficial because if they are somehow attached to the class definition through ANNOTATION (Java) or ATTRIBUTES (.NET, more specifically the attributes defined in DATAANNOTATIONS namespace that is part of the framework) then validation services can be written as a generic mechanism to determine whether the entity is in a valid state or not.

Annotating parameters and properties can cover a number of cases in which one can declaratively specify what PRECONDITIONS must be met, however it cannot cover them all. Some more complex method may require preconditions that involve multiple parameters. One solution to this problem is to define a class that holds the parameters and ensures their validity. While this may work it also emphasizes class explosion, not to mention the issue of naming those classes.

When it comes to inheritance, there is no language support for annotations. One can derive, override a property and completely change the annotations that would break the LISKOV SUBSTITUTION PRINCIPLE. The compiler will not check anything whether the constraints follow the principle.

Having a DOMAIN SPECIFIC LANGUAGE that covers the features that current implementations lack to be able to design by contract in any of the popular programming languages could prove beneficial. The language itself does not have to be complex and it revolves around predicates. The main requirement is to determine whether a set of constraints is less or is more restrictive than another set of constraints or whether such a relation exists. For instance, having a constraint for an integer to be

less than 5 and another to be greater than 6 then neither constraint is less or more restrictive than the other, one cannot substitute the other either.

Tools can be written to generate source code in other programming languages such as C#, JAVA or C++ that have checks for the PRECONDITIONS and POSTCONDITIONS. It would be rather rudimentary as an appropriate approach would be to follow the method implementation using the provided PRECONDITIONS and demonstrate that the POSTCONDITIONS always hold thus avoiding their check.

In the worst case the language mean to complete existing languages in order to be able to DESIGN BY CONTRACT becomes a GENERAL PURPOSE LANGUAGE and one can write entire applications within it. It would mean a great deal of work; however, compatibility would be less than an issue. One such language is EIFFEL.

While all seems to be in favor of DESIGN BY CONTRACT, there are some drawbacks. As always one must trade something to gain something else. In this case it is development time and maintenance time for more clear and robust code. In some cases, this technique may not prove beneficial, for instance when writing user interface controls, it could become difficult to specify through contracts how the interface should look like. Other cases would be when querying a database. Without some sort of unification or bridging of contracts between an SQL database or any other data storage system and the software system being developed the method invoking a SQL stored procedure has to trust that it works accordingly or it could validate the result. Sadly, not always can the result be validated as a SQL stored procedure can have side effects on multiple SQL objects.

1.3 Error Reporting

As constraints are validated it would be useful to know which, if any, constraints are not matched. Besides this the one defining them must have a way of returning an error message that details what is wrong to at least have a clue on how to mitigate the issue.

There are more ways to handle errors, one of them is to immediately interrupt the execution of the program by throwing an exception. This is most appropriate for methods, when the provided parameter does not match the constraints then there is not much the method can do therefore its execution should be interrupted and the caller should be informed of the issue through an exception. The error message specified by the constraint would be returned through the exception message.

In case of entities where data is provided through properties rather than parameters, having such a strict validation mechanism can become useless. It is possible to have entities that define constraints on one property in terms of another. The entity may require to go through an invalid state to get to a valid one. Having a more permissive validation policy can help in this case.

Invoking a validation service explicitly for a given entity to tell whether it is valid or not can be beneficial especially when the user is prompted to enter data to construct an entity. This way a full list of errors can be obtained and shown to the user rather than the first error and to get the next one the user has to resubmit the form.

In applications developed today, one cannot go ignorant about globalization at all. Most application are available in multiple languages. This may prove problematic if the error message for a

set of constraints is hardcoded as it will be available only in one language. Specifying error messages should not be about providing a constant string but rather an expression that evaluates to a string, this way the expression can be a reference to a resource from where the error message is loaded in one language or another depending on the user settings.

2 Conclusions

Having language support for DESIGN BY CONTRACT can be beneficial to the developer as constraints are expressed and carried out throughout class hierarchies without breaking SOLID principles. The resulting code would be of better quality as the validation of arguments would not be placed right before the implementation and part of the method body as if they were one and the same thing.

At least having a well-defined and robust language for expressing constraints can benefit not only the developers when they write code but also help communication between developers and domain experts. As part of the UBIQUITOUS LANGUAGE the definition of constraints can be understood by domain experts and they can assess whether it is correct or not and remove any doubt from the developers whether the provided information is accurate enough.

Validation is part of any software, with DESIGN BY CONTRACT and the ability to specify whether constraints are implicitly or explicitly checked one can write code that is more concise and can use the validation service to report back to the user what is wrong about the data they just introduced.

While great the use of DESIGN BY CONTRACT may be limited as connecting with various external services such as a SQL Server cannot guarantee their outcome and one may need to write extra validation code to ensure the results.

3 Future research

Future research will consist of comparing the tradeoffs between integrating a domain specific language into an existing programming language (C#), creating a whole new language that has built-in support for DESIGN BY CONTRACT (COMMON LANGUAGE SPECIFICATION compliant) and creating a compiler for an existing language that has built-in support for DESIGN BY CONTRACT (EIFFEL) that can be used by .NET COMMON LANGUAGE RUNTIME and thus any other .NET programming language (such as C#).

Besides having language support, another area of interest is to research how design by contract can be applied for USER INTERFACE design. How would user controls and related types benefit from such a paradigm if at all.

Last but not least it is of interest to research how DESIGN BY CONTRACT can interoperate with external services that do not have such features.

4 Bibliography

[1] B. Meyer, Touch of Class: Learning to Program Well with Object and Contracts, 1st ed., Springer-Verlag, 2009.

[2] B. Meyer, Object-Oriented Software Construction, 2nd ed., Prentice Hall, 1997.

[3] E. Gamma, R. Helm, R. Johnson and J. Blissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1st ed., Addison-Wesley Professional, 1994.

[4] B. H. Liskov and J. M. Wing, Behavioral Subtyping Using Invariants and Constraints, CMU technical report, 1999.

[5] M. Fowler and R. Parsons, Domain Specific Languages, 1st ed., Addison Wesley, 2010.

[6] E. Evans, Domain Driven Design: Tackling Complexity in the Heart of Business Software, 1st ed., Addison-Wesley, 2003.