Using MDE with the Clean Architecture

Sobhan Yassipour Tehrani[1], Kevin Lano[2]

[1]University College London, Gower Street, London, UK
[2]King's College London, Strand, London, UK

Abstract

The 'clean architecture' is a set of principles for software architecture, which aim to reduce the effort required for software maintenance and evolution. However it can require more initial effort, documentation and coordination within a development process, and hence is a challenge for agile developments. In this paper we describe how the use of model-driven engineering (MDE) can facilitate the application of the clean architecture principles and hence reduce the effort needed to employ them.

Keywords

Clean architecture, Model-driven Engineering, Agile development

1. Introduction

Agile development has become one of the most widely-used software development approaches in practice [4]. However, the focus of agile development upon coding rather than documentation, and upon rapid software production for immediate needs, creates challenges for the application of software architecture principles such as the clean architecture [8]. In particular, it is difficult to maintain architectural models, and to keep them up-to-date with code [2], [10]. In an agile context, frequent requirements change can lead to many changes to architectures, based on the requirements. We consider that one approach to address this conflict is to adopt an *agile MDE* approach [1], whereby development effort is focussed at the system specification level, with code artefacts being automatically synthesised from the specification. This facilitates rapid change to both specifications and code, and ensures that consistency is maintained between the specification and code, and between different code artefacts.

In Section 2 we review the clean architecture concepts, in Section 3 we describe the relation of MDE and the clean architecture, and describe detailed technical combination of MDE and the clean architecture in Sections 4 and 5.

2. Clean Architecture Concepts

The clean architecture [8] is a set of software design principles and practices oriented towards improving software maintainability and evolvability, and it focusses upon restricting the *dependencies* between software components, where a component X depends on component Y if X refers to Y (or to a class, interface or operation contained in Y).

The core principle of the clean architecture is the *dependency rule* DR:

> *Platform-specific components can depend on platform-independent components but not conversely.*

The dependency rule helps to insulate the relatively stable core business components of a system from the more frequently-changing components such as data persistence and UI elements. Thus the system becomes more maintainable and evolvable because the effort involved in technology changes is reduced.

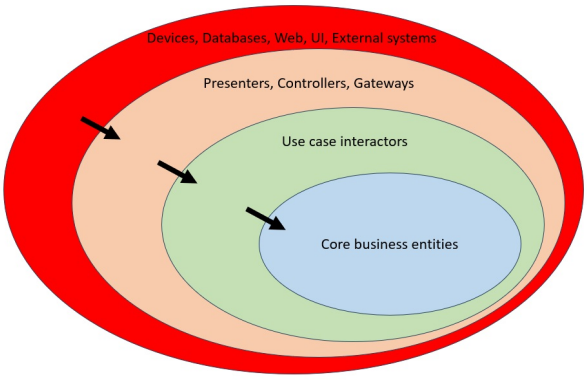

Figure 1: Clean architecture system organisation and permitted dependencies (From [8])

Figure 1 shows the clean architecture organisation of systems as layers of components each at the same level of abstraction from specific technologies. At the highest level of abstraction are components representing the

AMDE 2023: Agile Model-driven Engineering Workshop, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. K. Lano, H. Alfraihi, S. Rahimi and J. Troya, 20 July 2023, Leicester, UK.

sobhan.tehrani@ucl.ac.uk (S. Y. Tehrani); kevin.lano@kcl.ac.uk (K. Lano)

core business entities, which are independent of technologies and independent of the use cases of a particular application. For example, in any hotel reservation system we could have classes $Hotel$, $Customer$, $Reservation$, representing these concepts. The next layer consists of components for the business operations – components which implement the system functionality of the use cases of the particular application. These depend on the core business entities, but are independent of particular UI/data storage technologies. For example, a hotel reservation application could have operations to create a new reservation for a customer, or to update or cancel a reservation. The algorithms for these procedures reside in components termed the use case 'interactors' for such room reservation operations.

The next layer consists of intermediary components which link the business-specific parts of the system to UI and data storage technologies. Typically there are data-access objects (DAOs), otherwise known as 'Gateways' or database interfaces, to manage a specific data storage technology such as a relational database or cloud datastore. There are also components which manage UI interaction: Controllers, which control the UI navigation sequence and implement UI logic, and Presenters, which convert business data into specific formats (such as web page content) for display via particular UI technologies.

Finally at the outermost level of the system there are the platform-specific components such as relational databases, web pages and device drivers.

2.1. The Clean Architecture: SOLID principles

The clean architecture defines several principles of good component design and architecture design, which encompass many different kinds of system. The acronym SOLID is used for the following five principles of component design.

SRP: Single responsibility principle This principle states that *each component should have only one reason to change.* A more explicit statement is that *each component should provide services for a single actor.* SRP aims to avoid the situation where a component has overloaded responsibilities, trying to perform too wide a range of capabilities, for different clients, and hence becomes excessively large and complex and expensive to maintain.

OCP: Open-closed principle This principle states that components should be changed by adding new capabilities, and not by modifying existing capabilities. This provides a strong assurance of backwards compatibility for existing clients of the component, whilst retaining some flexibility to enhance and extend the component for new services and clients.

LSP: Liskov substitution principle This principle is a strong constraint on how superclasses and subclasses can be related [7]. Effectively it says that all subclass behaviour should satisfy the superclass specification.

ISP: Interface segregation principle This principle states that components should not depend on components or interfaces that they don't use: each such dependency causes wasted effort and resources when maintaining and deploying the system, and complicates the build and deployment processes.

DIP: Dependency inversion principle This principle states that a high-level client should not depend on a low-level supplier, because of the dependency rule. Instead, provided and required interfaces $ProvI$, $ReqI$ of supplier and client components are introduced, so that the dependencies are as follows: (i) from the client implementation to its required interface $ReqI$, (ii) from the implementation of the supplier upon $ProvI$, and (iii) *from* $ProvI$ *to* $ReqI$. The final dependency occurs because $ProvI$ needs to provide all the operations required by $ReqI$, in order that the assembly connection is valid. Figure 2 shows a UML component diagram of this structure.

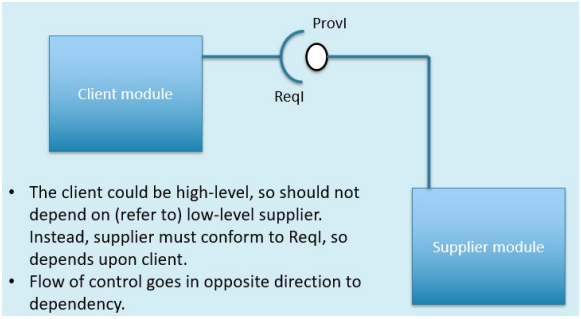

Figure 2: Client-supplier relationship with dependency inversion

Dependency inversion occurs if we place the required interface $ReqI$ into the client module, and the provided interface $ProvI$ into the supplier: the supplier module then has a dependency to the client. The dependency runs in the opposite direction to the calling relation: even though the client calls the supplier, no name of any supplier element occurs in the client.

This mechanism does not need to be used between every pair of client/supplier modules, only in situations where the client is at a higher abstraction level than the supplier, for example, when the client is a use case

interactor and the supplier is a gateway or presenter in the terms of Figure 1.

The clean architecture also states the following principle for the overall structure of an architecture:

- **ADP: Acyclic dependencies principle:** There should be no cycles in the component dependency graph.

3. MDE and Software Modelling

Model-driven engineering was established at around the same time as agile methods, in the late 1990's and early 2000's, and has also had extensive impact on the practice of software engineering, although it has been less widely adopted than agile methods.

Software models have several purposes:

- To facilitate communication and understanding about the system within the development team, and between the team and stakeholders
- To explore alternative designs
- To perform formal analysis, e.g., to check that clean architecture principles are being followed
- As a starting point for automated code generation.

An advantage of software modelling is that issues and alternatives for building the system can be explored before detailed coding begins. For architectural design, UML component diagrams such as Figure 2 provide a view of the possible ways in which a system can be decomposed into components, and these components assembled into the system as a whole. Issues such as the need for dependency inversion between two components can be recognised and resolved at this level of abstraction, prior to coding the components (Section 4).

Another use of models for software architecture is to *derive* an outline architecture from early-stage models such as business concept models and use cases, using these high-level models to infer what components, interfaces and connections are needed in the architecture of the system [3].

Finally, from detailed models, the code of a system can be generated, including interfaces, components and dependencies conforming to a particular architectural style and to the clean architecture principles (Section 5).

Version 2.2 of the AgileUML MDE toolset[1] supports the definition of UML component diagrams, the derivation of classes and interfaces from these diagrams, and checks for conformance to the SRP, ISP, DR and ADP clean architecture principles.

[1] github.com/eclipse/agileuml

4. Analysis of Clean Architecture Principles at the Specification Stage

The use of MDE for system specification enables the early detection of architectural issues and problems, such as violations of the clean architecture principles. Figure 3 shows one form of analysis which can be performed: the usage relations between use cases and the data of class diagram classes can be automatically identified and graphically presented (green arrows denote read access and red arrows write access).

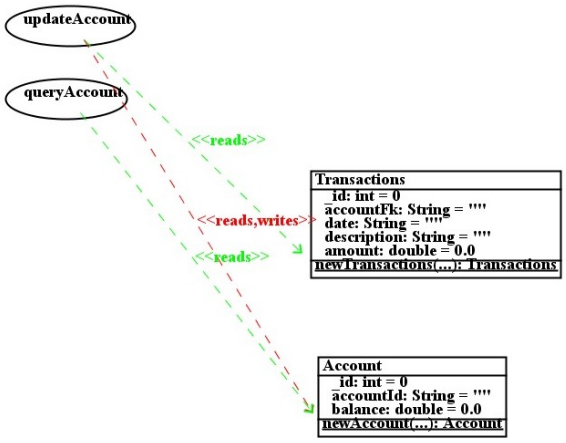

Figure 3: Use case read and write dependencies to class diagram

This form of analysis can detect violations of SRP, because these occur when two different use cases with different actors both access the same class (Figure 4). In this example, the $Staff$ actor performs the $updateAccount$ use case, whilst the $Customer$ actor performs the $queryAccount$ use case.

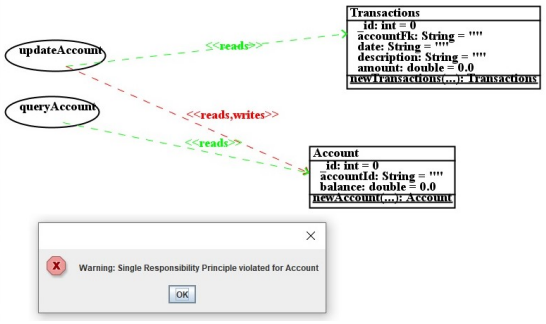

Figure 4: SRP violation example

Table 1
Clean architecture checks

Principle	*Check*
DR	Warning issued if a platform-independent class is directly linked by an association to a platform-dependent class
SRP	Warning issued if a class is referenced by two use cases which have different actors
OCP	Warning issued when an operation definition is changed, if there is any call to the operation in the system
LSP	Informal or formal verification can be carried out based on superclass and subclass operation specifications [5].
ISP	(i) Check if there are excessive ($\geq$ 20) numbers of operations in an interface. (ii) Check if operations of supplier objects are called: if no operation of referenced class S is called within client C, then S can be removed as a supplier of C (Figure 5).
ADP	(i) Identify calling cycles between operations of different classes; (ii) Identify reference cycles between classes.

Table 1 summarises the software modelling checks which can be performed for the clean architecture principles. Settings for thresholds, such as the limit of number of operations in an interface, can be modified by changing the default values given in $TestParameters.java$.

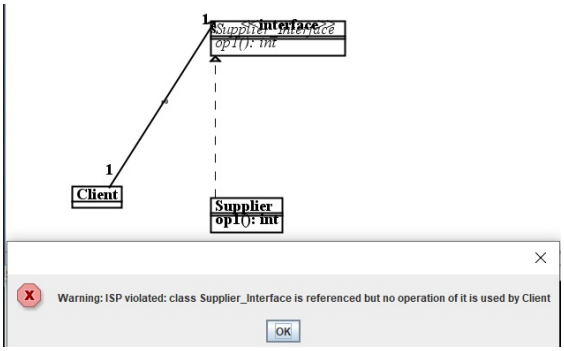

Figure 5: ISP violation example

With regard to the dependency inversion principle, component diagrams can be used to generate class diagram elements which have the appropriate interfaces and relationships to support this. Figure 6 shows the generation of a class diagram (in the top pane) from a component diagram (lower pane).

To check the dependency rule, the AgileUML tool identifies platform-specific and independent components, and checks if platform-independent components directly refer to platform-specific components (Figure 7). A class is considered platform-specific if it directly utilises files, processes, Excel functions, databases, sockets or internet requests (library classes $OclFile$, $OclProcess$, $Excel$, $OclDatasource$ and $SQLStatement$).

5. Generation of Clean Architectures from Design Models

MDE can also be used to generate the executable code for applications, together with components, interfaces and connections conforming to particular architectural styles or principles, in particular to the clean architecture. This can substantially reduce the manual effort required to write code and organise it into appropriate architectures. In particular, there may be some necessary duplication of code within an architecture:

- In the UI tier, validation checks on the input data for a use case are typically based on the *business rules* for the use case, which are also represented in the business tier. E.g., that a room reservation must have start date before or equal to the end date.
- Passing data between components in value objects – the value objects often have data based on that of the core business entities.

By using automated code generation from design models, the necessary validation beans and value objects can be generated from the same model entity/operation definitions as the core business components. This ensures consistency between the different artefacts, and reduces coding effort. The design model in this respect serves as the 'single source of truth' for the application code, and should be maintained during system evolution.

As an example, consider a $Reservation$ business entity from the hotel room booking system. A detailed specification of this entity could be:

```
class Reservation
{ attribute reservationId : String;
```

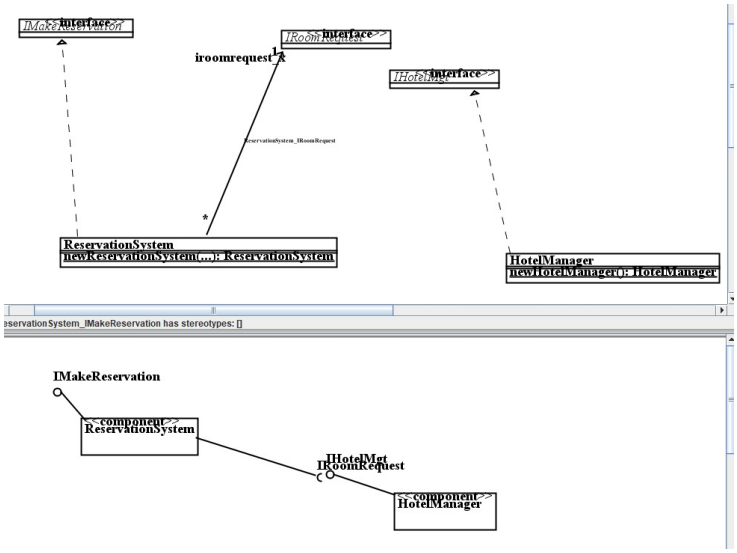

Figure 6: Class diagram synthesis from component diagram

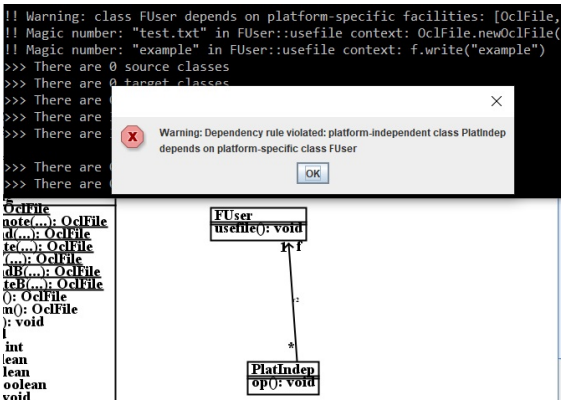

Figure 7: Dependency rule violation example

```
  attribute startDate : String;
  attribute endDate : String;
  invariant startDate <= endDate;

  reference hotel : Hotel;
  reference customer : Customer;
  ....
}
```

The invariant states that the reservation interval must be non-empty. Dates could be represented in the format "YYYY:MM:DD", so that string comparison corresponds to chronological order (e.g., "2023:12:31" $<$ "2024:01:01"). This invariant should be checked for the input data of $makeReservation$ and $updateReservation$ use cases, and this could be done by a validation bean $ValidateReservation$, called from a UI tier component such as a Controller (Figure 8). In addition, the invariant needs to be checked in the database interface or Gateway for data storage of $Reservation$ instances. Thus the same semantic rule appears in widely-scattered places in the application.

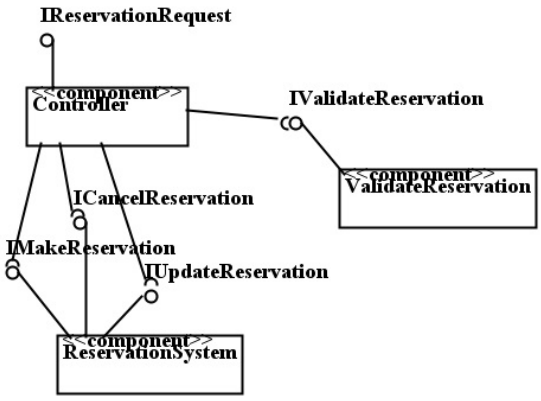

Figure 8: Room booking system UI tier/business tier

AgileUML can generate web-based applications, desktop applications and iOS and Android mobile apps from UML/OCL specifications [5]. Each of these implementations may involve the production of multiple inter-related components at different levels in the clean architecture hierarchy. In the case of SwiftUI mobile app synthesis

[6], we use the model-view-view model (MVVM) app architecture, with the following specific components:

- Business entity classes $E.swift$ are generated for each business entity E.
- The use case interactor component $ModelFacade.swift$ implements all application use cases, and is generated from the specifications of these cases.
- SwiftUI views for each use case are generated from the use case specification.
- Data access objects/gateway components $DAO\_E.swift$ are generated for each business entity E.
- Value objects/data transfer objects $eVO.swift$, $ucVO.swift$ are generated for each business entity and for each use case.
- A database interface for local SQLite database storage is generated for the local persistent business entities.
- A cloud database interface for remote data storage is generated for remote persistent business entities.

It would be very difficult to manage this coding and to maintain consistency between different components manually. Instead, any changes to the system requirements can be carried out by manual specification changes, which are then automatically propagated to consistent changes to the code of all components.

One problem with the use of MDE is the integration of auto-generated code with manually-written and maintained code. This integration can be facilitated by separating MDE-generated and manual code into separate components, which only interact via interfaces. For example, in an embedded system, the low-level device drivers could be manually coded and maintained, whilst the main control component would be auto-generated from design models. When a change is needed to the strategy for code-generation, this change should be made to the MDE code generators, not by manual modification of the generated code.

6. Related Work

An MDE approach designed to generate mobile app architectures satisfying the clean architecture is described in [11]. However this is specific to an Android and firestore implementation, whereas our approach is generally applicable to any domain. Discussion of the clean architecture in relation to mobile app patterns is given in [9]. They propose a systematic use of the clean architecture, which could be achieved using the MDE approach described here.

Conclusions

In this paper we have identified the practical application of Model-Driven Engineering (MDE) techniques in enforcing clean architecture principles in architectural design. The automated generation of components, interfaces, and connections through MDE enables developers to maintain architectural integrity and coherence throughout the software development process. Additionally, the utilization of MDE techniques facilitates adaptability and agility in the face of changing requirements.

References

[1] H. Alfraihi, K. Lano, *The integration of agile development and MDE: a systematic literature review*, Modelsward 2017.

[2] E. C. Arango, O. L. Loaiza, *SCRUM Framework Extended with Clean Architecture Practices for Software Maintainability*. In: Silhavy, R. (eds) Software Engineering and Algorithms. CSOC 2021. Lecture Notes in Networks and Systems, vol 230. Springer, Cham. https://doi.org/10.1007/978-3-030-77442-4_56

[3] J. Cheesman, J. Daniels, *UML Components*, Addison-Wesley, 2000.

[4] R. Hoda, N. Salleh, J. Grundy, *The Rise and Evolution of Agile Software Development*, IEEE Software, July 2018.

[5] K. Lano, *Agile Model-based Development using UML-RSDS*, CRC Press, 2016.

[6] K. Lano, S. Kolahdouz-Rahimi and L. Alwakeel, *Synthesis of mobile applications using AgileUML*, ISEC 2021, pp. 1–10, 2021.

[7] B. Liskov, J. Wing, *A behavioral notion of subtyping*, ACM Trans. Program. Lang. Syst. 16(6), 1994, pp. 1811–1841.

[8] R. Martin, *Clean Architecture*, Prentice Hall, 2018.

[9] R. Nunkesser, *Choosing a global architecture for mobile applications*, TechRxiv, https://doi.org/10.36227/techrxiv.14212571.v1, 2021.

[10] F. Gomes Rocha, S. Misra, M. Soares, *Guidelines for Future Agile Methodologies and Architecture Reconciliation for Software-Intensive Systems*, Electronics 2023, 12, 1582, 2023.

[11] D. Sanchez, A. Rojas, H. Florez, *Towards a clean architecture for Android apps using model transformations*, IJCS, vol. 49, no. 1, 2022.