Aggregate Decoupling Pattern

CHRISTOPHER HARTLEY, Cisco Systems

Domain Driven Design Aggregates provide a useful way of improving a model's coupling and cohesion.

In some cases, more advanced techniques are needed to further decouple the aggregate definitions.

This paper documents a method of decoupling aggregates by denormalizing associations to other aggregates and redundantly storing information.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—Patterns

General Terms: Knowledge, Pattern, Architecture, Information, Instance, Metamodel

Additional Key Words and Phrases: Aggregate

ACM Reference Format:

Hartley, C. 2022. Aggregate Decoupling Pattern. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 29 (October 2022), 15 pages.

1. BACKGROUND

The concept of Aggregates was proposed by Eric Evans in his book "Domain Driven Design" (DDD).

> *"Cluster the ENTITIES and VALUE OBJECTS into AGGREGATES and define boundaries around each. Choose one ENTITY to be the root of each AGGREGATE, and control all access to the objects inside the boundary through the root. Allow external objects to hold references to the root only. Transient references to internal members can be passed out for use within a single operation only. Because the root controls access, it cannot be blindsided by changes to the internals. This arrangement makes it practical to enforce all invariants for objects in the AGGREGATE and for the AGGREGATE as a whole in any state change."* [Evans]

Aggregates provide a useful architectural restriction that improves fine-grain coupling and cohesion in a practical way.

Note that Aggregates are not the same as UML Aggregation, even though the names are similar.

In his book, Eric Evans uses several different examples. One recurring example used is that of a cargo transport model that is built up throughout the book to highlight various DDD concepts.

Figure 1 below is a modified version of the model in the book, combining the many examples into one diagram and renaming the classes for clarity.

Note that in the diagrams in this paper, Aggregate Roots are shown with yellow fill and aggregate Leaves with green fill. Aggregate roots are tagged with an <<Agg_Root>> stereotype and Leaves with a <<Agg_Leaf >> stereotype. Associations within an aggregate boundary are tagged <<Agg_Within>> and Associations that cross Aggregate boundaries are tagged as <<Agg_Between>>.

From figure 1, it is clear that there are two layers in the example model, the left layer (column) is the cargo layer and the right is the underlying transport layer. The question is, what is the best place to interconnect the two layers, from the lowest pink one being the most detailed up to the higher grey one (CargoLegOnTransportVoyage).

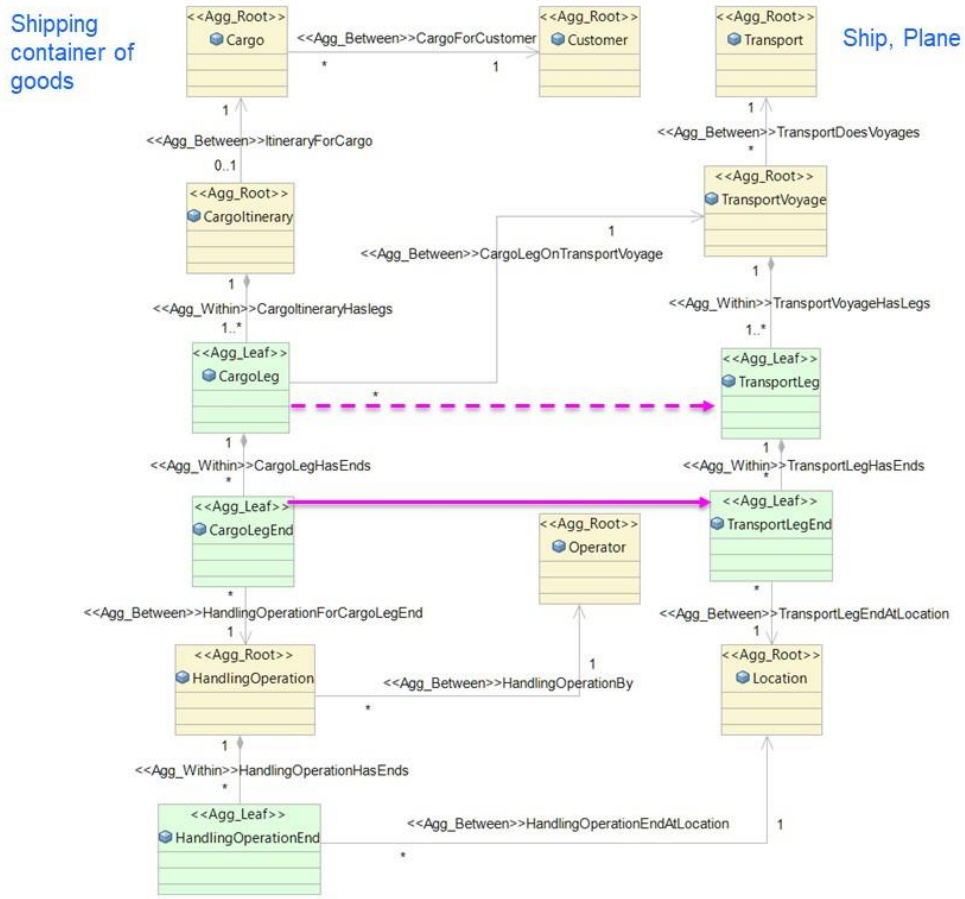

Figure 1 – Diagram based on Evan's Cargo Examples

The "Aggregate Decoupling Pattern" in this paper is based on an example slide "Fewer Assumptions" from a Domain Driven Design training course held by Eric Evans that the Author attended.

Note in Figure 1, that the two pink possible associations break Evan's Aggregate rule "*Allow external objects to hold references to the root only*" and would require those referenced classes to be made Aggregate Roots to be compliant with the Aggregate rules.

This paper will cover the tradeoffs in decoupling, aggregate definition and complexity in choosing the best interconnection points between Aggregates in cases where the obvious connection is problematic.

2. INTENT

In most cases the definition of Aggregate boundaries and the interconnections between them is very straightforward for well-defined models. In some cases, however, it may not be possible to achieve a good and simple solution, while conforming to the Aggregate rules.

This *Aggregate Decoupling Pattern* can be used to reduce the coupling between selected Aggregates, where a model created using a domain modelling approach is determined to have too much coupling.

Note that there are a large number of correct Aggregate model structures [Hartley2021], so for this pattern only one simple structure is shown.

The intent of the pattern is to replace an association from an external Aggregate into a Leaf with one that references the Root instead, by moving the terminating end from a Leaf to the Root.

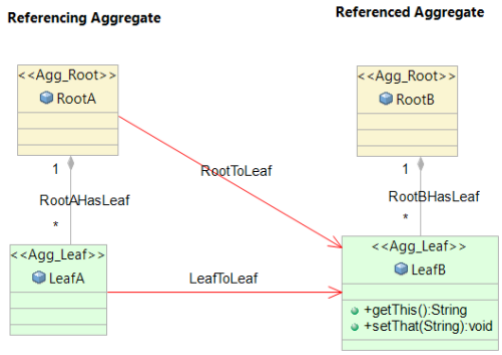

Figure 2 – Incorrect Aggregate Coupling

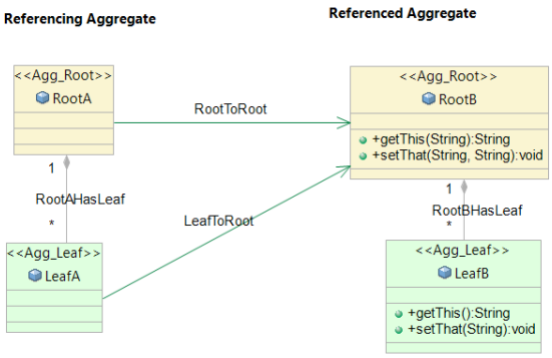

Figure 3 – Correct Aggregate Coupling

The key here is the asymmetry of the <<AggBetween>> association between the two aggregates (due to the navigability). Since Roots manage all their leaves the rules are different at each end.
For example, an association from $Leaf_{A3}$ to $Leaf_{B3}$ would become as shown in figure 4 below.

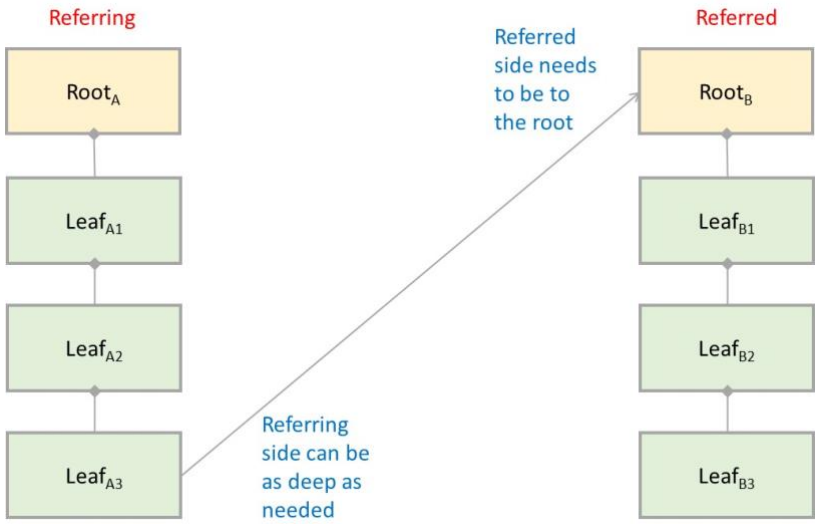

Figure 4 – One-way asymmetry

If a both-way association between $Leaf_{A3}$ to $Leaf_{B3}$ was required, then the result would be as shown in figure 5 below.

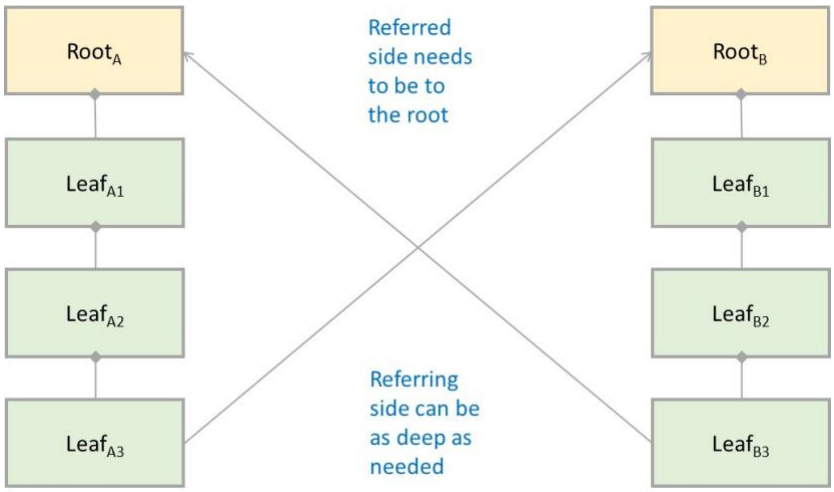

Figure 5 – Both-way asymmetry

3. MOTIVATION

Reducing the coupling between Aggregates helps to make them more cohesive and hence better suited to modern modular, message-based software architectures.
The other motivation is to reduce the cost of maintaining a reference between aggregates over time. The cost increases as the stability of the information in the referenced aggregate decreases.

4. APPLICABILITY

Use this approach when modelling using Aggregates and the desired level of decoupling has not been achieved. The most likely issue will be that the desired coupling requires an association with navigability into a Leaf class from another Aggregate, breaking Evan's Aggregate rule "*Allow external objects to hold references to the root only*".

5. PARTICIPANTS

The participants will be two Aggregate definitions, with a referencing Aggregate requiring a reference into a Leaf of the referenced Aggregate.

6. COLLABORATIONS

If an Aggregate needs to interact with a Leaf in another Aggregate, then it will need to do this through the other Leaf's Root class.
The referencing Aggregate can keep information about the referenced Leaf in lieu of a direct reference, but there will need to be a way of keeping this denormalized information up to date. It makes sense to minimize the amount of data about the referenced Leaf, to just an identifier if possible (and with a value as stable as possible too).

7. CONSEQUENCES

Storing remote information rather than a reference may lead to a loss of data integrity. This means that there needs to be some form of notification to keep the information up to date.

An Aggregate Root needs to have a global identifier, but an Aggregate Leaf only needs a local identifier (so it can be referenced within the Aggregate). This means that if another aggregate stores a local identifier, then the form and value of the local identifier is exposed externally. If the local identifier value can change then some change notification mechanism needs to be in place.

8. IMPLEMENTATION

Replacing a direct reference to a Leaf class with one to its Aggregate Root, requires the Root to implement the Leaf method signatures and delegate to the Leaf methods as appropriate (while also ensuring that the Aggregate consistency is maintained).

9. KNOWN USES

The cargo and cargo transport layers have a lot in common with communication networks, for example an IP packet layer transported by another layer like Ethernet frames.

10. RELATED PATTERNS

None known at this point in time.

11. A WORKED EXAMPLE

The following example builds up the intent of the pattern:

- The example is for air travel between airports
- It is simplified to highlight the issues of concern, and is not intended to be implemented
- It is not intended to cover all possible fault cases (like unscheduled stops for fuel, aircraft crashes, flight abandonment and then returning to the same gate …)
- The initial scenario is:
  - An airport can have one or more buildings (called concourses) that each have one or more departure / arrival gates
  - Aircraft fly on flight paths from one gate to the other, allowing passengers to depart from one gate and arrive at another

For the Flight, we will use the EdgeEnd Class pattern [Hartley2020]. The pattern chosen is not really important for this presentation, this is just to clarify the resulting model structures.

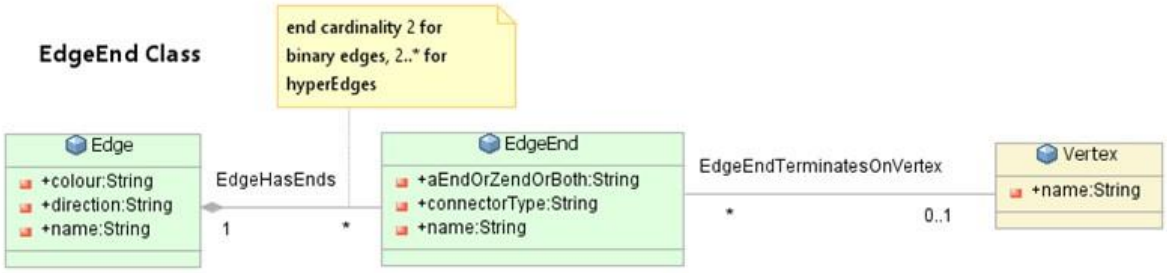

Figure 6 – Graph Pattern used for the Flight-Airport Example

A simple ‘obvious’ starting point model is shown below, that covers the problem definition.

Figure 7 – The Initial Example Model

This UML model matches the simple picture below.

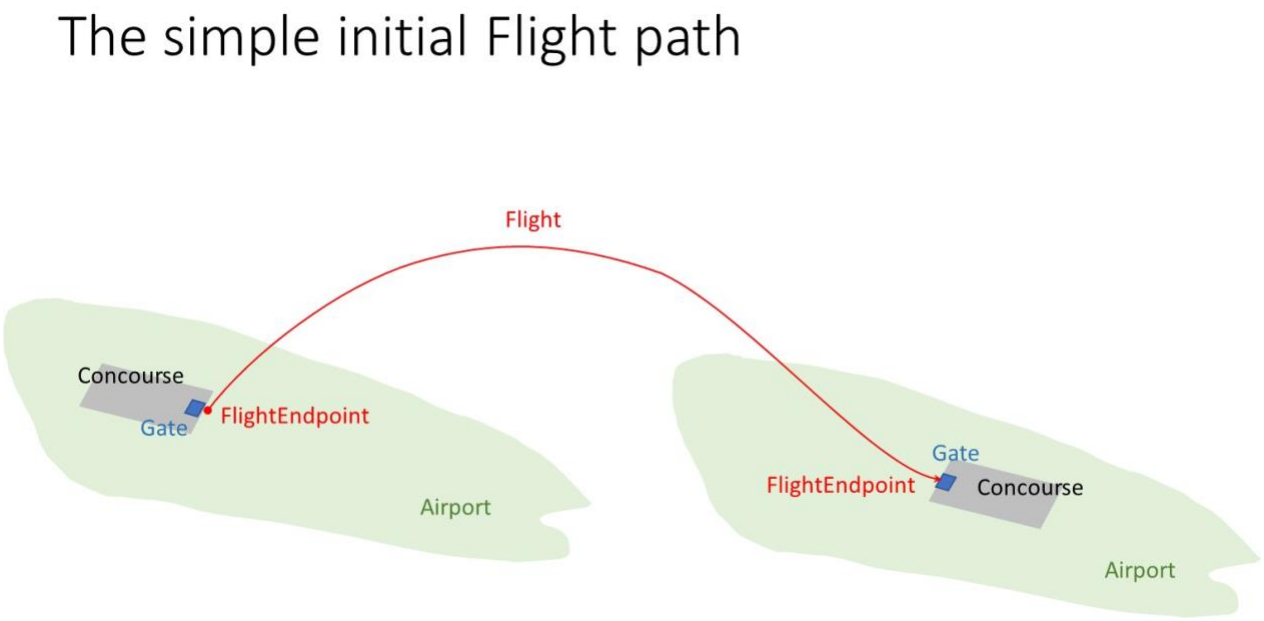

Figure 8 - Initial Example Diagram

After adding the Aggregate tagging and coloring, the diagram becomes as shown below.
Now we can see the benefit of using Aggregates, as it shows an issue with the FlightEndpoint reference into Gate.

We could just make Gate an Aggregate Root, but let's think about the actual problem a bit more !

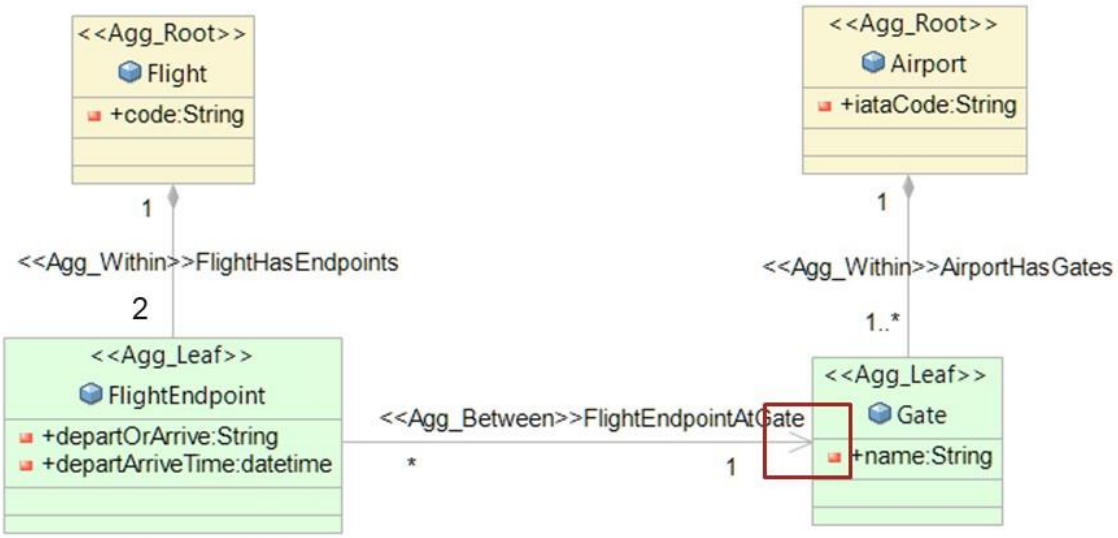

Figure 9 – The initial Example Diagram with Aggregate Tags

Thinking a bit more about the problem:

- A Flight may be defined a year or more in advance
- The gate allocations may only happen an hour or two before departure
- On long flights, often the arrival gate is not known at departure time !

It seems that the Flight has too much coupling to Gate:

- The Gate doesn't really need to know about Flights, all it needs to know is its own state – is the door open or not, is the aircraft bay occupied or not

Perhaps the solution could be to make Gate a Root of its own Aggregate, but:

- Even if we make Gate a Root, we still have the temporal issue.
- Making FlightEndpointAtGate.gate optional to avoid the temporal issue causes even more issues as we really need to record the start and finish airports when a Flight is created
- If Gate is now optional, we could then add a second association (FlightEndpointAtAirport) to Airport and only add the association instance to Gate when the gate is known, but that still requires Gate to be a root.

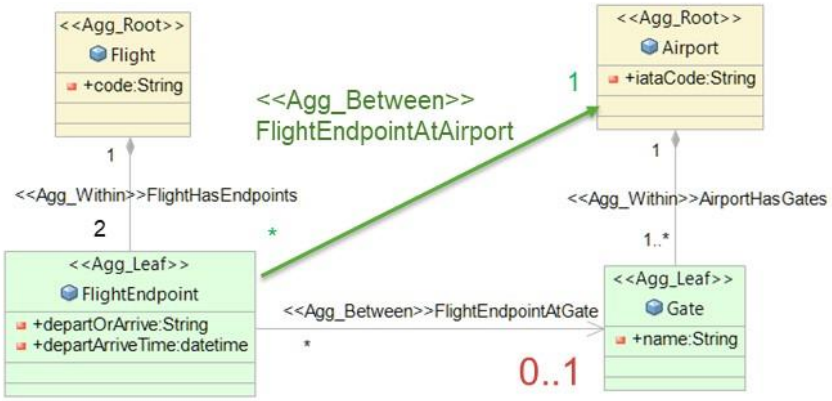

Figure 10 – Adding a Second Association between Flight and Airport

This two-association solution with Gate made a Root, could be an acceptable solution in many (monolithic) implementations where coupling and cohesion is not important. In a more modern modular distributed implementation, converting natural Leaves to Roots causes other issues (discussed later).

Airlines are responsible for flights and Airports for gates, so we should reduce the coupling across this boundary.

- By moving the association end to Airport, we can decouple the Flight from the Gate
- FlightEndpoint.gateName can now be optional, and is not part of the 'unique key' of FlightEndpoint
- gateName has been chosen rather than an internal Gate identifier as it seems to make more sense to use a public identifier. Note that gate allocations are made less than 24 hours out and the gate local identifier is unlikely to change in that time. Historical gate allocations don't really need updating either. If a "gate renumbering" change was underway, the Airport class would be the best placed anyway to enforce the changeover and could allocate old and new values based on the changeover versus flight times.

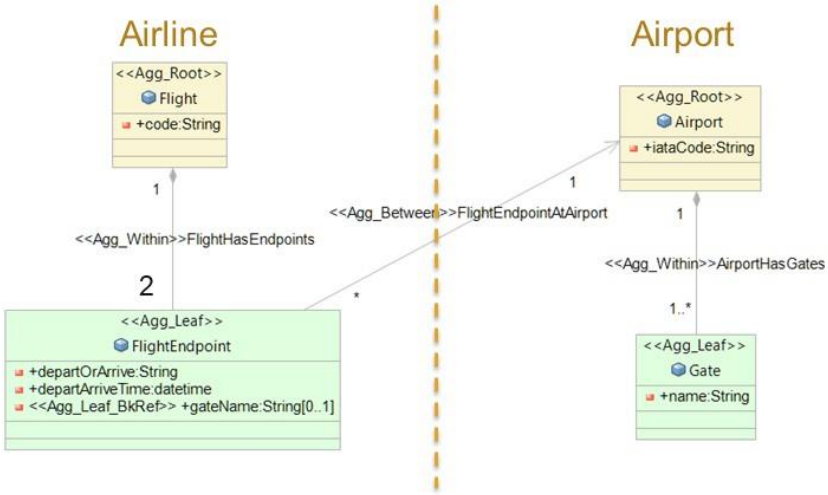

Figure 11 – Decoupling Flight from Airport

In figure 12 below, this simple example is extended to make it more realistic in covering real-world issues:

- By adding details of the runway that the flight uses / is assigned to
- Airport gates are per concourse, so a bit more structure can be added to the model to see how to cope with referring to a leaf that is a number of levels under a root
- By adding the customer side, where customers book seats on flights. Once again, initial allocations are to a flight and the seat allocations happen much later. It is likely that the airline flight planning systems and the customer booking systems would be somewhat decoupled – exchanging messages to keep in synch. Note that this is a common situation where the initial allocation is quite general and is refined over time.
- One thing to consider is that if we directly couple the trip booking leg to the flight seat, we can ensure that a seat is not double booked. The use of aggregates highlights that it is the responsibility of the aggregate root, Flight in this case, to make sure that it is internally consistent, rather than trying to code it in the static structure of the model.

In the diagram below, the red <<Agg_LeafRef>> arrows are an attempt to show the indirect references using UML by using dependency relationships.

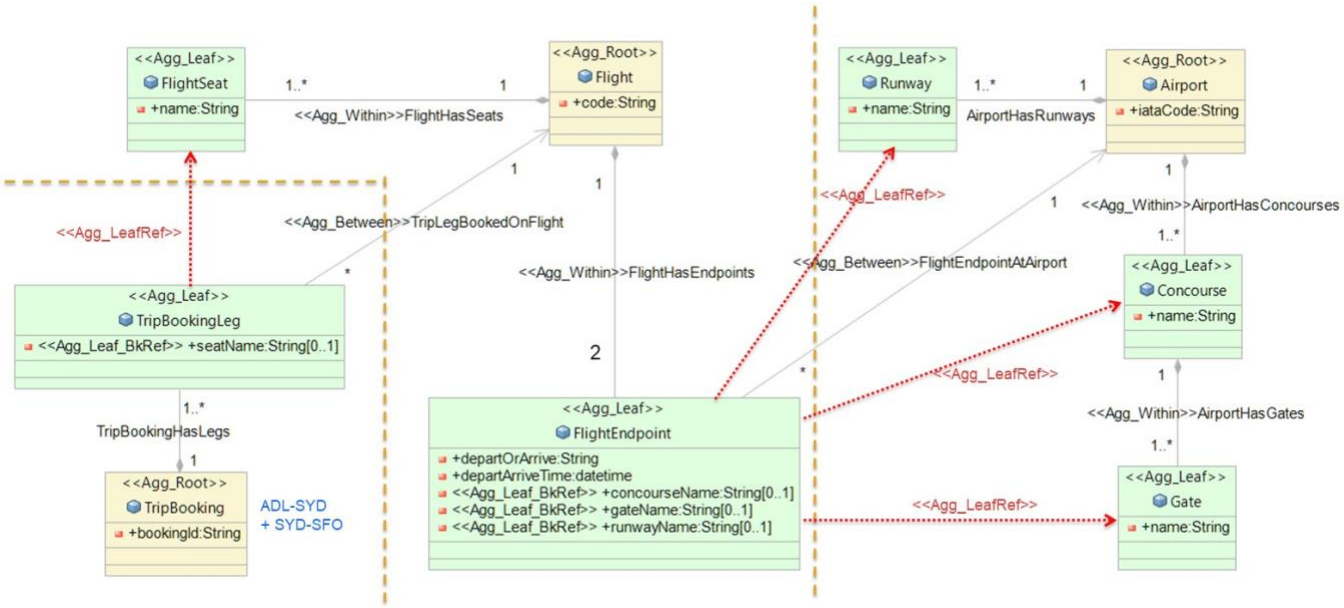

Figure 12 – Expanded Example

Figure 12 shows three examples of Aggregates referencing external Leaves that we said are quite rare, but it should be remembered that:

1. this example is designed to focus on these cases
2. if the model was fully developed then there *may* not be any more of these cases and they *may* then constitute a small percentage of all of the associations

This is also quite fundamental to the concept of modelling, creating a useful representation rather than trying to produce an exact representation.

If the model is to be implemented on a distributed message-based architecture, then the choice of message boundaries becomes important. Messaging within a monolith has little impact as the objects can directly call each other's methods. Messaging and consistency is a big concern in distributed systems and means that the modelling strategy needs to be more careful about coupling and cohesion levels.

If the defined aggregates align with the natural problem space 'unit of consistency' [Vernon] then passing an Aggregate instance per message makes sense.

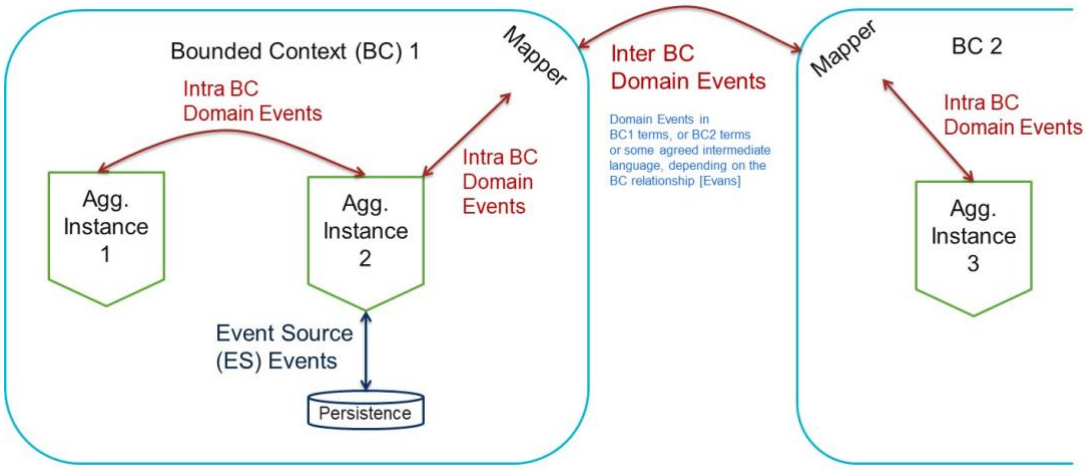

Figure 13 – A Possible Event Architecture

In a possible messaging architecture using an event log like Apache Kafka between the Airline and Airport flight scheduling systems:

- The use of aggregates clarifies the domain events to pass around in Kafka
- A *TripBooking* can send requests to a *Flight* and receive an allocated or not response
- A *Flight* can request a takeoff or landing slot at an *Airport* and receive an allocated or not response
- As the *Flight* progresses, it can send update messages to the customer support system and *Flight* can receive gate and runway update messages from the *Airport*
- The software intelligence is in the Aggregate Root classes and they message each other and ensure the consistency within their Aggregate boundaries
- An *Airport* object needs to remember the *Flight* to *Gate* and *Runway* allocations it has already made. Rather than relying on a shared monolithic database with hidden coupling, *Airport* can use the Kafka log as its shared store with *Flight*

Note that depending on the implementation, the objects may message each other via some other intermediate layers, depending on any system or microservice boundaries between them.

•

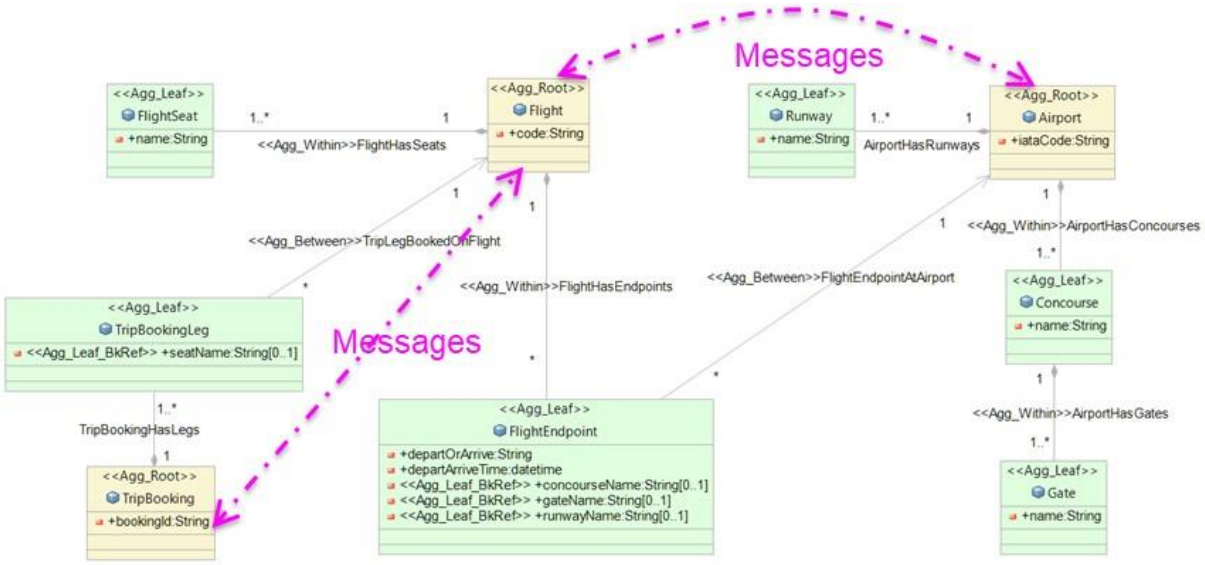

Figure 14 – How the model could relate to a messaging architecture

Now we have all the pieces in place for a worked example to show how the information evolves over time. Note that there could be a number of systems that share copies of similar (overlapping) information.

Here we will assume that we are looking at the problem from the point of view of an airline flight management system interacting with an airport terminal management system using asynchronous events.

Airport, Runway, Concourse and Gate information should be well known ahead of time and relatively static. Any Airport structural work will be known in advance and planned for.

Flight and Flight endpoint should also be well defined and planned in advance, but may be subject to change (or cancellation) on the day of the flight.

What is less well defined is the relationship between flights and the gate allocations.

A few hours before a flight is due to depart, the Airline flight system requests airport terminal management system to be allocated a gate for departure.

It is likely that the gate allocation will have a status relating to how definite it is. An aircraft would stay on the taxiway until the gate allocation is confirmed and the gate is free (no aircraft parked outside).

As the departure process progresses, the gate status will change (allocated, confirmed, occupied, free), our aircraft will park at the gate, open the door, load the passengers, close the door, then leave. The Airline flight system and airport terminal management system can share Flight and Gate status updates respectively. The only issue is if the gate is changed during the process, the references need to be updated so that our system responds to the correct messages.

It seems the advanced decoupling pattern could also be applied to other areas where the state is refined over time, such as:

- Hotel room bookings, where the customer books a type of room and at check-in is assigned to an actual room
- Order Fulfilment / Shipment to associate to Order rather than OrderLineItem

12. CONCLUSION

This paper showed that when applying Aggregates to a model, that in some cases where the model breaks the Aggregate rules by having an external reference to an Aggregate Leaf, that changing the model structure by moving the referencing association may be appropriate.

A worked example then explored application of the pattern and the thought processes involved.

While the *Aggregate Decoupling Pattern* may not be one that is used every day, it is a useful tool to have available to use when needed.

13. APPENDIX A – AN ANALYSIS OF GENERAL AGGREGATE BASED REFACTORING

In his book [Refactoring], Martin Fowler documents a number of changes that can be made to object-oriented code.

The refactorings have names like:

- Extract Superclass
- Replace primitive with Object
- Separate Query from Modifier

This section explores if, in the same fashion to Fowler's book, we can define a list of Aggregate specific refactorings. This list is not aiming to be exhaustive, but covers the ones that have been seen and used in existing Aggregate tagged models.

The starting assumption is that there is an existing model and that every concrete class has been tagged as an Aggregate Root or Leaf

13.1 Extract a Leaf from an Aggregate (and convert to a Root)

This is a simple case of 'moving the Aggregate boundary'.

13.2 Move Leaf Between Aggregates

This is another simple case of 'moving the Aggregate boundary'.

13.3 Split Class into Root plus Leaf

If an Entity **cannot** be clearly defined as a Root or Leaf, then this suggests that there are some major issues with the model. The suggested solution is to model the Root case and Leaf case separately and then look at what level of merging of the two models is possible. It is not really possible to be more explicit in this case, and is best to let the problem domain guide the solution.

13.4 Split Association into separate Agg_Within plus Agg_Between Associations

If a single association covers both the Agg_Within and Agg_Between cases, then split it into two separate associations and check the navigability and end multiplicities for each case separately:

- Reduce the navigability of each replacing association, if possible
- Reduce the multiplicity of each replacing association end, if possible (to make the multiplicity more restrictive, changing the lower bound from '0' to '1', and changing the upper bound from '*' to '1' if possible)

For a Bothway Agg_Between Association:

- Reduce the navigability of the association, if possible
- Consider which end needs the association visibility as part of its (semantic) definition
- If both sides need visibility, then perhaps there is an intermediate Entity that needs to be explicitly represented (reified)

For a Both-way Self-Join Multi-Use Association, this can be considered to be the worst-case scenario and is worth exploring in detail how to solve this problem by breaking it up into steps.

1. Split the association into Agg_Between and Agg_Within cases
2. Determine the correct multiplicities and navigabilities for each case
3. If there are now any Leaf - Between - Leaf associations, then see if the Aggregate Decoupling Pattern can be applied, by moving the terminating end up to a Root

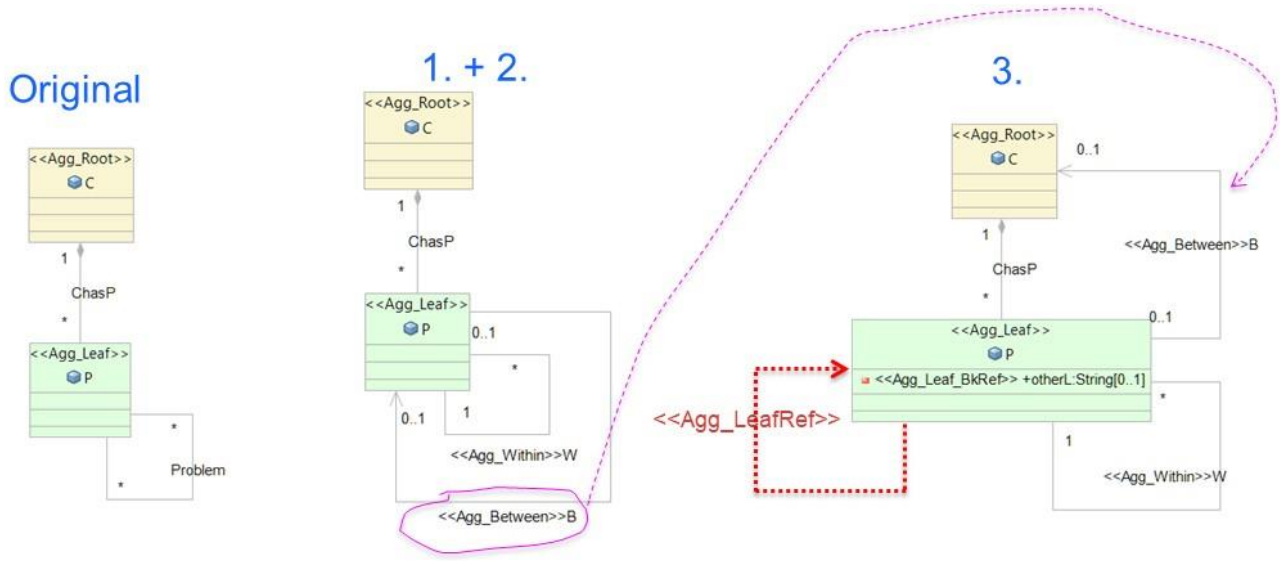

Figure 15

For an Inherited Both-way Self-Join Multi-Use Association (Composite Pattern), this is a special case of the previous special case:

– 0. The first step is to split the association back to the longhand non-inherited form as shown in figure 16 [Hartley2021]
– 1. Then proceed as before with steps 2 and 3 for each of the associations from 0.

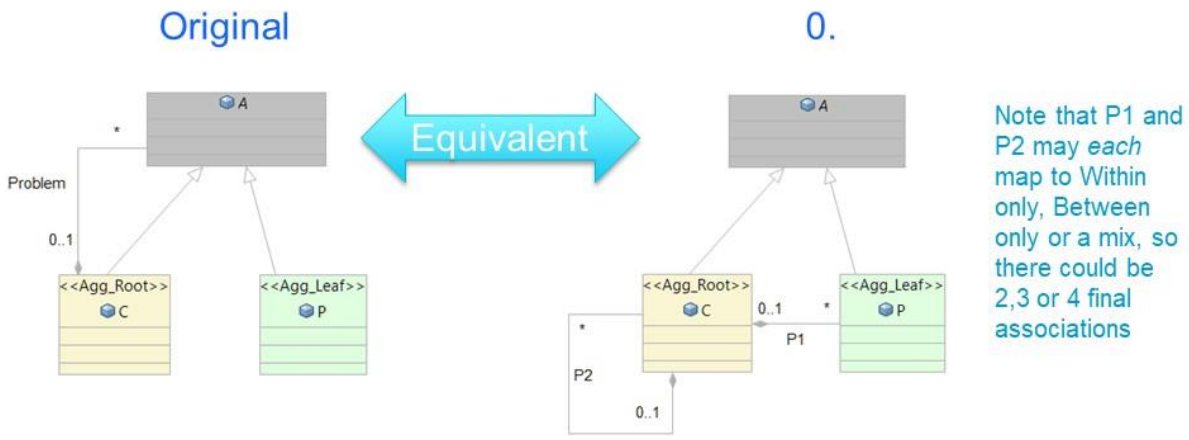

Figure 16

For an Inherited Both-way Self-Join Multi-Use Association (Non-Composite), This is a special case of the original special case:

– 0. The first step is to split the association back to the longhand non-inherited form as shown in figure 17 [Hartley2021]
– 1. Then proceed as before with steps 2 and 3 for each of the associations from 0.

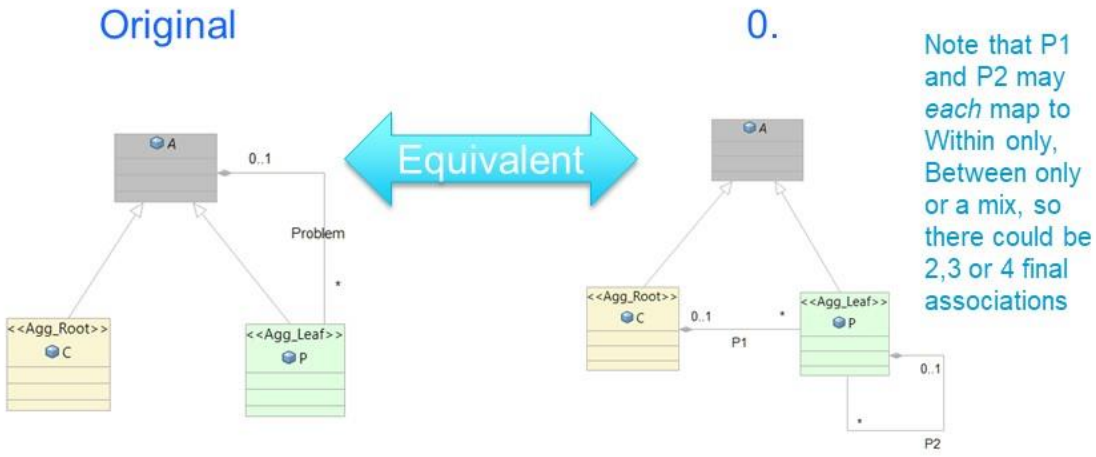

Figure 17

Note that these examples are not claimed to be exhaustive, and other similar cases may exist. They should be able to be treated in a similar fashion.

ACKNOWLEDGMENTS

Thanks to Nigel Davis for doing the first review of this paper.
Thanks to Eric Evans for allowing me to use his training slide as the basis for this paper.
Thanks to Y C Cheng for shepherding this paper.

REFERENCES

[Evans] Domain-Driven Design : Tackling Complexity in the Heart of Software by Eric Evans
ISBN: 0321125215

[Fowler-Aggregate]
https://martinfowler.com/bliki/DDD_Aggregate.html

[GoF] Design Patterns: Elements of Reusable Object-Oriented Software
by Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides
ISBN-0201633612

[Refactoring] Refactoring: Improving the Design of Existing Code
By Martin Fowler
ISBN-10: 0134757599

[Vernon] Implementing Domain-Driven Design by Vaughn Vernon
ISBN: 9780321834577

[Hartley2020]
Hartley, C and Davis, N. 2020. An Abstract Graph Pattern Collection. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 27 (October 2020), 21 pages.

[Hartley2021]
Hartley, C. 2021. Recursive Patterns for an Aggregate World. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 28 (October 2021), 40 pages.

GLOSSARY

ACRONYM	**DEFINITION**
UML	Unified Modelling Language
ONF	Open Networking Foundation https://www.opennetworking.org/
DDD	Domain Driven Design
DDD Aggregate	Aggregate is a pattern in Domain-Driven Design. A DDD aggregate is a cluster of domain objects that can be treated as a single unit. [Fowler-Aggregate]
DDD Bounded Context	Is an architectural boundary of a model and its concepts and vocabulary.

Received June 2022