Project Creation Architecture Strategy

The TTA Tools system will use the Anti-corruption Layer strategy to interface with multiple instrument-specific conceptual models used for scheduling and executing observations. This strategy creates an isolating layer to provide clients with functionality in terms of their own domain model. Internally, the layer can translate in both directions as necessary between the two models but in practice we expect the translation to be one way - from TTA Tools to the instrument-specific systems.

Project Creation Architecture Analysis

1. GBO, NRAO-VLA, and NRAO-VLBA have their own distinct models and related systems for scheduling and executing observations.
2. The ngVLA model for observation scheduling and execution is not known.
3. The GBO, NRAO-VLA, and NRAO-VLBA systems are implemented with different technologies and are maintained by different staff.
4. Coordinating planning and development between GBO and NRAO-VLA or NRAO-VLBA has historically been difficult for a variety of reasons that are not expected to change.
5. The current funding trajectory for GBO does not support the idea that the GBT software staff will once again become part of NRAO-DMS.
6. For these reasons, the Project Creation part of the TTA Tools conceptual architecture is based on the assumption that the GBO, NRAO-VLA, and NRAO-VLBA models will remain bounded contexts. This means DMS will maintain the scope of a particular model as a bounded part of the TTA Tools software system within which a single model will apply and will be kept as unified as possible.
7. Therefore, DMS will make no attempt to create one coherent model that encompasses all instruments for scheduling and executing observations. Consequently, in the GBO context an idea like ‘session’ can continue to mean something different from the NRAO-VLA context of ‘session’.
8. Nevertheless, the TTA Tools system must “support the creation of observing projects for each allocation request with positive disposition in a format appropriate for each facility.”
9. **The problem then is how to best deal with multiple models.**
10. There are six patterns which cover a range of strategies for relating two models that can be composed to encompass an entire enterprise[1].
11. The patterns are: Shared Kernel, Customer/Supplier Development Teams, Conformist, Anti-corruption Layer, Separate Ways, and Open Host Service. Each is explained in the following pages.

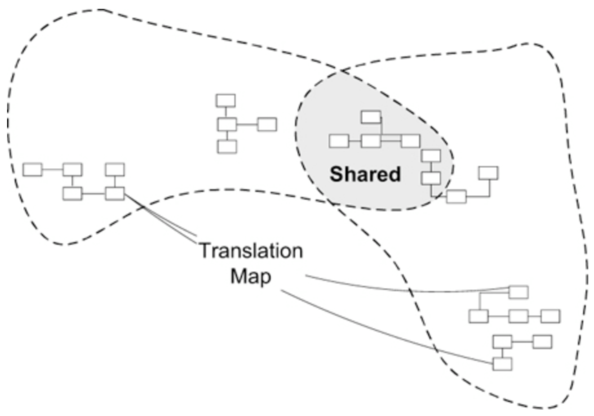
[1] All of these ideas and the text describing the patterns belong to “Domain-Driven Design:...”, Eric Evans, Ch 14

12. Conformist and Separate Ways can be eliminated immediately because they inherently do not support #8.
13. Shared/Kernel and Customer/Supplier Development Teams are likely not good options due to #4.
14. Open Host Services is not practical because that strategy implies some shared model vocabulary. As a result, the other subsystems become coupled to the model of the Open Host and other teams are forced to learn the particular dialect used by the Host team. #1 suggests this would not be a successful approach.
15. *By elimination, Anti-corruption Layer is the chosen strategy. The core idea of this strategy is to create an isolating layer to provide clients with functionality in terms of their own domain model. The layer talks to the other system through its existing interface, requiring little or no modification to the other system. Internally, the layer can translate in both directions as necessary between the two models but in practice we expect the translation to be one way - from TTA Tools to the instrument-specific site systems.*
16. #15 assumes the TTA Tools system captures all the information necessary to schedule and execute observations at the different instrument sites and will accommodate ngVLA in the future.

Strategies for Relating Two Models

Shared Kernel

1. **Designate some subset of the domain model that multiple teams agree to share.** Of course this includes, along with this subset of the model, the subset of code or of the database design associated with that part of the model. This explicitly shared stuff has special status, and shouldn't be changed without consultation with the other team.Integrate a functional system frequently, but somewhat less often than the pace of Continuous Integration within the teams. At these integrations, run the tests of both teams.



Customer/Supplier Development Teams

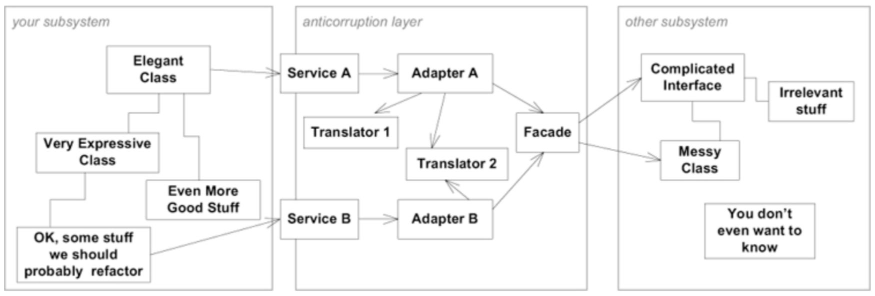
1. Often one subsystem essentially feeds another; the “downstream” component performs analysis or other functions that feed back very little into the “upstream” component, and all dependencies go one way. The two subsystems commonly serve very different user communities, who do different jobs, where different models may be useful. The tool set may also be different, so that program code cannot be shared.
2. **Establish a clear customer/supplier relationship between the teams.** In planning sessions, make the downstream team play the customer role to the upstream team. Negotiate and budget tasks for downstream requirements so that everyone understands the commitment and schedule.
3. Jointly develop automated acceptance tests that will validate the interface expected. Add these tests to the upstream team’s test suite, to be run as part of its continuous integration. This testing will free the upstream team to make changes without fear of side effects downstream.
4. During the iteration, the downstream team members need to be available to the upstream developers just as conventional customers are, to answer questions and help resolve problems.
5. Automating the acceptance tests is a vital part of this customer relationship.
6. The relationship must be that of customer and supplier, with the implication that the customer’s needs are paramount. This situation is in contrast to the poor-cousin relationship that often emerges, in which the downstream team has to come begging to the upstream team for its needs.

Conformist

1. When two teams with an upstream/downstream relationship are not effectively being directed from the same source, a cooperative pattern such as Customer/Supplier Teams is not going to work.
2. When two development teams have an upstream/downstream relationship in which the upstream has no motivation to provide for the downstream team’s needs, the downstream team is helpless. Altruism may motivate upstream developers to make promises, but they are unlikely to be fulfilled. Belief in those good intentions leads the downstream team to make plans based on features that will never be available. The downstream project will be delayed until the team ultimately learns to live with what it is given. An interface tailored to the needs of the downstream team is not in the cards.
3. **Eliminate the complexity of translation between Bounded Contexts by slavishly adhering to the model of the upstream team.** Although this cramps the style of the downstream designers and probably does not yield the ideal model for the application, choosing Conformity enormously simplifies integration. Also, you will share a Ubiquitous Language with your supplier team. The supplier is in the driver’s seat, so it is good to make communication easy for them. Altruism may be sufficient to get them to share information with you.

Anti-corruption Layer

1. New systems almost always have to be integrated with legacy or other systems, which have their own models. Translation layers can be simple, even elegant, when bridging well-designed Bounded Contexts with cooperative teams. But when the other side of the boundary starts to leak through, the translation layer may take on a more defensive tone.
2. When a new system is being built that must have a large interface with another, the difficulty of relating the two models can eventually overwhelm the intent of the new model altogether, causing it to be modified to resemble the other system's model, in an ad hoc fashion. The models of legacy systems are usually weak, and even the exception that is well developed may not fit the needs of the current project. Yet there may be a lot of value in the integration, and sometimes it is an absolute requirement.
3. **Create an isolating layer to provide clients with functionality in terms of their own domain model.** The layer talks to the other system through its existing interface, requiring little or no modification to the other system. Internally, the layer translates in both directions as necessary between the two models.
4. The public interface of the Anti-corruption Layer usually appears as a set of Services, although occasionally it can take the form of an Entity. Building a whole new layer responsible for the translation between the semantics of the two systems gives us an opportunity to re-abstract the other system's behavior and offer its services and information to our system consistently with our model. It may not even make sense, in our model, to represent the external system as a single component. It may be best to use multiple Services (or occasionally Entities), each of which has a coherent responsibility in terms of our model.
5. One way of organizing the design of the Anti-corruption Layer is as a combination of Facades, Adapters (both from Gamma et al 1995), and translators, along with the communication and transport mechanisms usually needed to talk between systems.



Separate Ways

1. We must ruthlessly scope requirements. Two sets of functionality with no indispensable relationship can be cut loose from each other.
2. Integration is always expensive. Sometimes the benefit is small.

3. **Declare a Bounded Context to have no connection to the others at all, allowing developers to find simple, specialized solutions within this small scope.**

Open Host Service

1. Typically for each Bounded Context, you will define a translation layer for each component outside the Context with which you have to integrate. Where integration is a one-off, this approach of inserting a translation layer for each external system avoids corruption of the models with a minimum of cost. But when you find your subsystem in high demand, you may need a more flexible approach.
2. When a subsystem has to be integrated with many others, customizing a translator for each can bog down the team. There is more and more to maintain, and more and more to worry about when changes are made.
3. If there is any coherence to the subsystem, it is probably possible to describe it as a set of Services that cover the common needs of other subsystems.
4. **Define a protocol that gives access to your subsystem as a set of Services. Open the protocol so that all who need to integrate with you can use it.** Enhance and expand the protocol to handle new integration requirements, except when a single team has idiosyncratic needs. Then, use a one-off translator to augment the protocol for that special case so that the shared protocol can stay simple and coherent.
5. This formalization of communication implies some shared model vocabulary—the basis of the Service interfaces. As a result, the other subsystems become coupled to the model of the Open Host, and other teams are forced to learn the particular dialect used by the Host team.