Agile/Evolutionary Data Modeling: From Domain Modeling to Physical Modeling

@scottwambler

I'd like to start with a few simple definitions:

- Data modeling is the act of exploring data-oriented structures.
- Evolutionary data modeling is data modeling performed in an iterative and incremental manner.
- Agile data modeling is evolutionary data modeling done in a collaborative manner.

This article effectively describes an evolutionary approach to data modeling, it is your choice whether you want to apply these techniques in an agile (highly collaborative) manner or not. I recommend that you take an Agile Model Driven Development (AMDD) approach.

Data modeling techniques are used in several ways -- domain data modeling (analysis), logical data modeling (detailed analysis), architectural data modeling, and physical data modeling (design). It is perfectly fine to use data models for several purposes, or different types of models for a similar purpose. For example, you can use data models, CRC models, UML class diagrams, and ORM diagrams for domain modeling; as Agile Modeling (AM) suggests, know Multiple Models so that you can and Apply the Right Artifact(s) for the situation. In this article I discuss a agile/evolutionary approach to data modeling.

Unfortunately traditional data professionals often prefer to work in a (near) serial manner : they'll create a mostly complete domain model, then perhaps create a logical data model (LDM) based on the domain model, and once that LDM is accepted create a physical data model (PDM) based on it. Although there is opportunity to update the models as a project progresses this is often a difficult and time consuming effort because the underlying assumption is that the database schema will be set very early in the project and be left alone. This is convenient for the data professionals because it streamlines their work but doesn't reflect the iterative and incremental (evolutionary) processes such as Scrum, or better yet Disciplined Agile Delivery (DAD) commonly followed by developers. This article shows how data professionals can easily adopt an evolutionary, and better yet agile, approach to data modeling.

Table of Contents

1. The Case Study

The purpose of this fictitious project is to build a Karate School Management System (KSMS) for a single dojo. We start the project by doing some initial modeling where we envision the initial requirements for KSMS, in the form of user stories, as well as the initial architecture. A user story is a reminder to have a conversation with a stakeholder, hence there are not a lot of details provided in the table. When we start working on a user story we work closely with the user, ideally getting them involved with the modeling effort via Agile Modeling's Active Stakeholder Participation practice. As we explore the requirement we capture UI-related ideas, business rules, and structural information (e.g. business entities, the relationships between them). We also implement the requirement, hopefully taking a TDD-based approach which enables us to do less detailed design modeling because the tests capture this critical information, in both our objects and the database. The system will be built using object technology (e.g. J2EE or C#) on the front end and relational technology (e.g. MySql, Oracle) on the back end.

Disclaimer

This article focuses on the data aspects of the system, which is only one of a myriad of issues which agile software developers must address. Therefore, I do not show the other artifacts (e.g. source code, class diagrams, architecture diagrams, ...) which we would create in parallel to the data models. Furthermore, I'll keep the data models relatively simple, leaving out details such as columns used to record the creation date of a row or the date it was last updated, so we can instead focus on the approach that I use to create the data models.

2. The Initial Requirements

Table 1 lists the user stories describe the initial usage requirements for KSMS. Our stakeholders have prioritized the requirements and the developers have estimated the effort to implement them. Based on the priorities and estimates we have assigned the requirements to 6 two-week iterations.

Table 1. Initial User Stories.

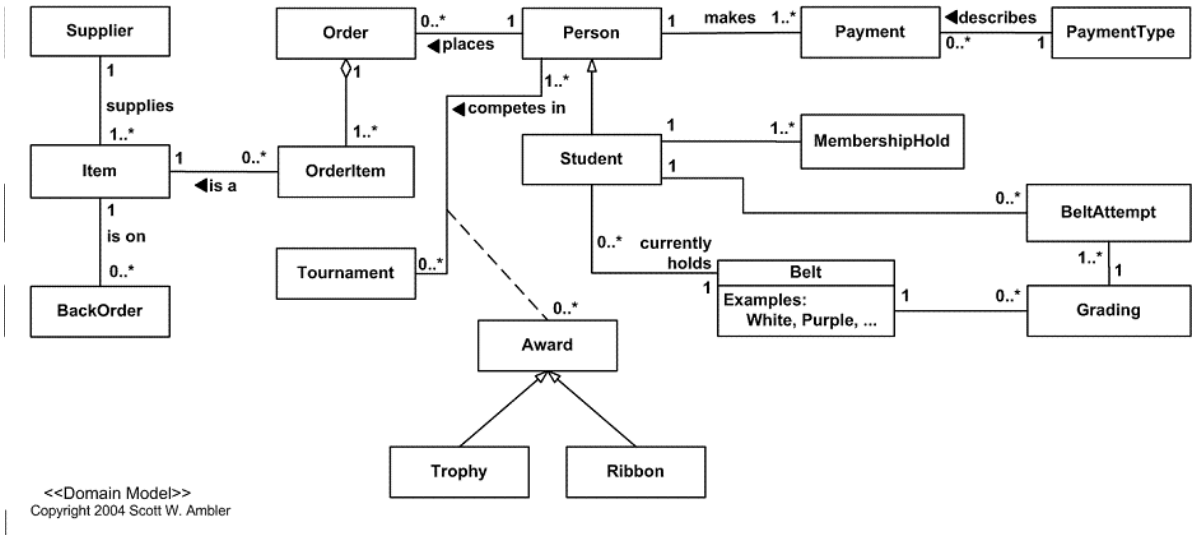
Iteration	User Stories

1	• Maintain student contact information • Enroll student • Drop student • Record payment
2	• Promote student to higher belt • Invite student to grading • Email membership to student • Print membership for student
3	• Schedule gradings • Print certificate • Put membership on hold
4	• Maintain product information • Sell product
5	• Print catalog of products • Order product for inventory • Order product for student
6	• Organize tournament • Enroll participant in tournament • Send out tournament announcement email to past participants • Print tournament announcement letters to past participants

3. The Initial Domain Model

Part of your initial modeling efforts, particularly for a business application, will likely include the development of a conceptual domain model. This model should be very slim, capturing the main business entities and the relationships between them. Figure 1 shows this model using UML data modeling notation (you can use any notation that you like when agile data modeling, I prefer UML). The initial domain model will be used to help guide both the physical data model as well as the class design, potentially captured via a UML class diagram (the class design is out of scope for this article).

Figure 1. The initial domain model.



The only supporting documentation which I would create for this model would be a definition of the entities, information I'd be tempted to capture in a glossary. I wouldn't bother identifying attributes for the entities at this time, this is information that is better captured in either the class schema or the database schema. Furthermore, I would draw a model such as this on a whiteboard to start out with, and would likely keep it on the whiteboard throughout the project. My experience is that a slim domain model such as this is a valuable asset to the project team, one that should be very easy to view (you should simply have to look up from your desk and view the shared whiteboards in your work room), to create (whiteboards are very inclusive), and to modify.

> Lesson #1: Agile data modelers travel light and create agile models which are just barely good enough. It's

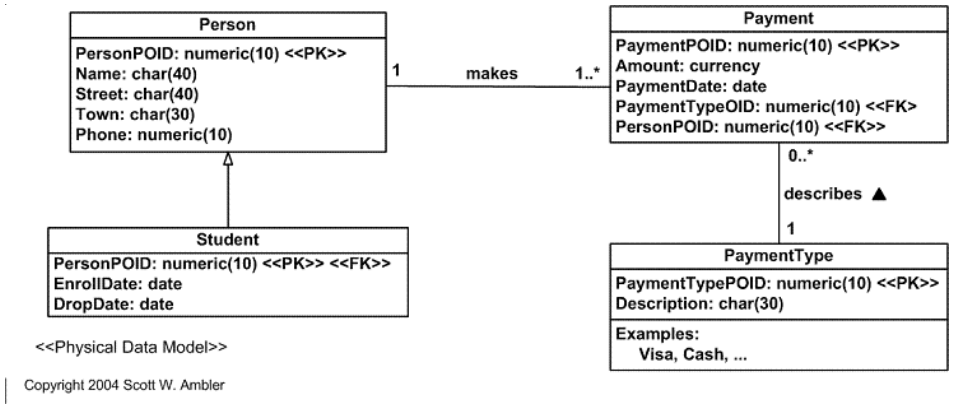
important to ask yourself: When is Enough Modeling Enough?

4. Iteration 1

There are four user stories to fulfill in this iteration: Maintain student contact information, Enroll student, Drop student, and Record payment. Therefore we need to do the work to implement those four, and only those four requirements. The first iteration of the KSMS physical data model (PDM) supports the critical functionality required to run the dojo - the management of basic student data and collection of money from them. When you take a look at the data model you see that we're not tracking the state/province that a person lives in. Because we're building for a single dojo, which is nowhere near the border, we can safely assume that everyone lives in the same province.

Lesson #2: Agile data models are just barely good enough for the task at hand. Agile developers solve today's problem today and trust that they can solve tomorrow's problem tomorrow, therefore at a later date if we need to support people living out of province then we'll add that functionality at that time.

Figure 2. The iteration 1 PDM.



All database changes are made in parallel to required code changes. My development partner, Beverley, and I worked together to evolve both the Java code and the database schema. It isn't enough to take an evolutionary approach to your database design, you also need to take a collaborative approach. We worked together, often pair programming, on the entire system. I generally took the lead on the database work and she generally took the lead on the Java work but the critical point is that we worked together to get the application built. One of us didn't design what needed to be done and hand it off to the other, a serial approach which risks communication errors. Nor did we go our separate ways and each do our own part of the work, a parallel approach which risks double work (both of us would have explored the same schema issues, her from an object point of view and me from the data point of view) and incompatible work (we could have easily made different schema design decisions).

Lesson #3: agile data modeling is both evolutionary and collaborative.

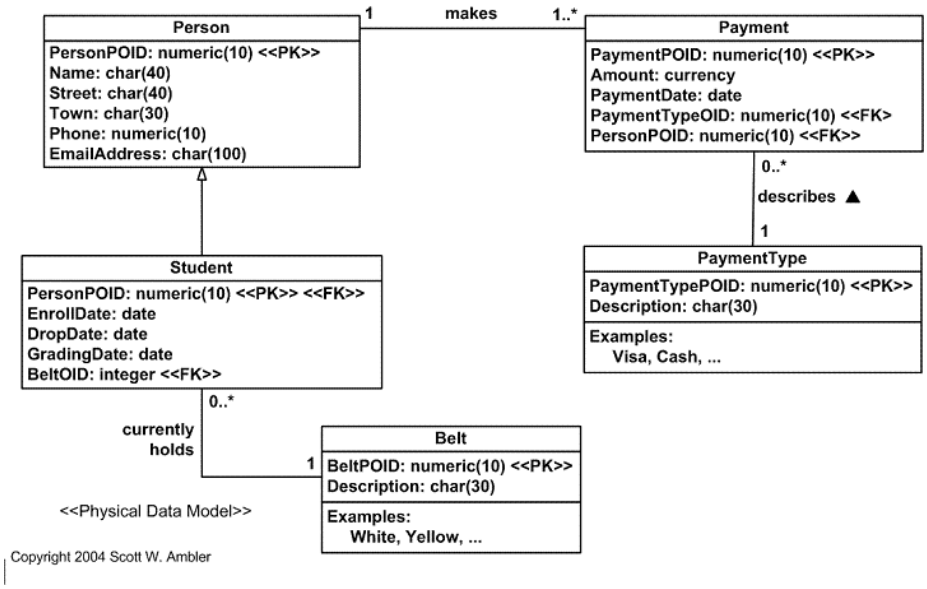
It's also important to note that we're following both an Agile Model Driven Development (AMDD) and a Test-Driven Design (TDD) approach. The topic of this article is agile data modeling, hence I will focus on AMDD concepts. However, it is important to recognize the importance of agile testing strategies in agile approaches. Regression testing is a critical success factor for evolutionary development; without a full regression test suite in place you can't safely evolve your work, including your database schema. Many agilists take a TDD-based approach to implementation where they write unit tests before they write their actual code. You can do this for both your object source code as well as for your database schema code (DDL, stored procs, ...) using common testing tools.

You can take any key strategy and use any notation that you like when agile data modeling. We've chosen to keep the key strategy simple, using surrogate values called persistent object identifiers (POIDs) which are unique across all records within the database. We could have used natural keys for many tables, or even just surrogate values unique only within each table, but POIDs seem to work incredibly well.

5. Iteration 2

There are four user stories to implement during this iteration: Promote student to higher belt, Invite student to grading, Email membership to student, and Print membership for student. This functionality has no significant overlap with the existing data model and as you can see in Figure 3 the changes were fairly straightforward. The *Belt* table and the *Student.BeltOID* column were added to support the ability to track which belt a student currently has. We also added the *Person.EmailAddress* column so we can email membership information to students and the *Student.GradingDate* column to track the last/next time a student grades for a new belt.

Figure 3. The iteration 2 PDM.



We could have added the *EmailAddress* column to *Student* instead of *Person*, but we're already joining these two tables to obtain student information and email addresses aren't just applicable to students alone. Are we overbuilding the system? Not really, because the application code is only using email addresses within the context of students which is exactly what the requirements call for. However, just because we don't want to overbuild my software doesn't mean we need to be stupid about the way that we model the data schema: we can look ahead a bit and organize the database schema so that it reflects the domain and not just the specific requirements of the application which we're currently supporting. However, we're doing this in such a way that we don't impact the application schedule and we're doing it in a collaborative manner with the developers (I'd discuss with them why I think that we should put *EmailAddress* in the *Person* table and verify that it doesn't impact them adversely). The implication is that we might not be able to do everything that we want to do right now but that we can do many of the easy things (such as introducing the *Person* table which is clearly not needed, yet, by the application).

I'd like to make two points about Figure 3:

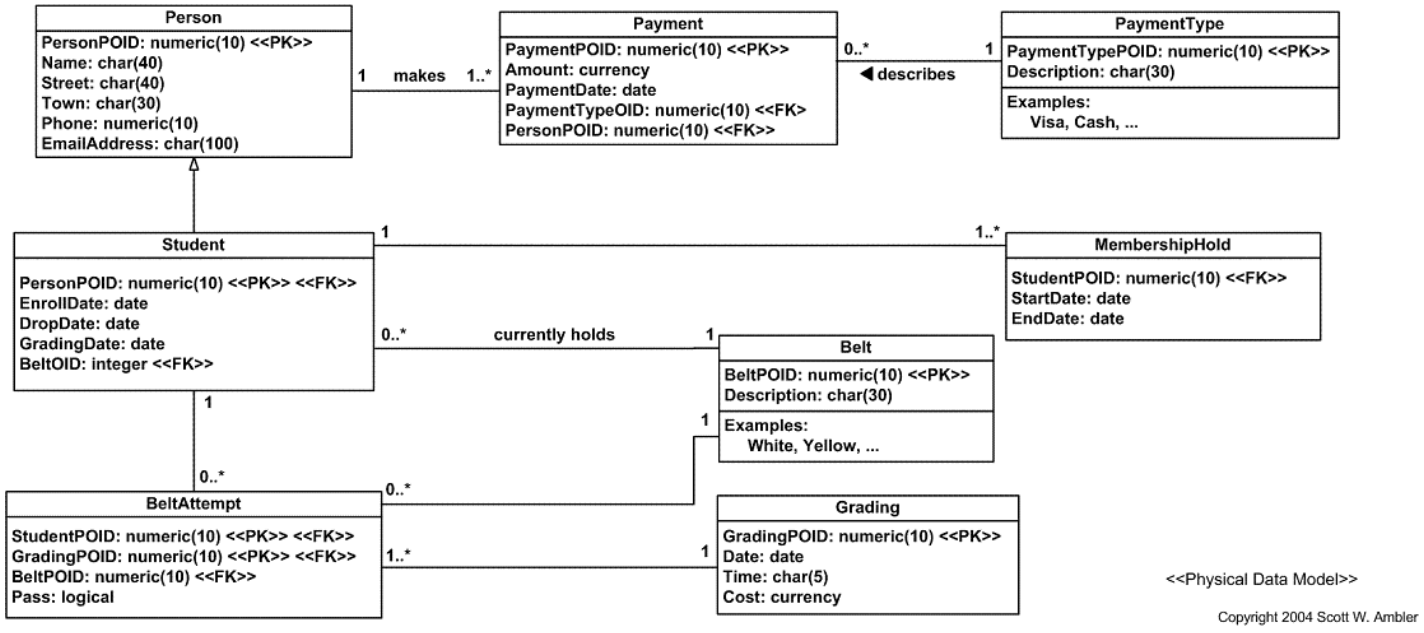
1. We've followed the Agile Modeling (AM) practice of Apply Modeling Standards and followed good naming conventions and even modeling style guidelines.
2. We're only tracking the current belt that the student has, not their entire history (e.g. we're not tracking when they earned their yellow belt, then their orange belt, and so on). Nor are we adding extra columns right now, just in case we might need them at some point in the future. Although this might be useful the reality is that we don't have a requirement to do this work and it would impact the application code because they'd need to overbuild. Remember, agile data models are just barely good enough.

Lesson #4: You can be agile yet still support the needs of the enterprise.
Lesson #5: Agile data models can and should follow your corporate standards.

6. Iteration 3

For this iteration we have three user stories to implement: Schedule gradings, Print certificate, and Put membership on hold. Sometimes students will stop training for awhile, it's quite common for people to go away on vacation during the summer, and the dojo will put their membership on hold so that they're not charged when they're not there. To support this functionality I added the *MembershipHold* table to keep track of when the membership was on hold, allowing the system to track the number of weeks used from a given membership (memberships are either 3, 6, or 12 months in length). To manage gradings we needed to add two new tables, *BeltAttempt* which tracks the belt a student is attempting during a given grading and *Grading* which tracks basic information about the grading. These tables were straight additions to the database schema.

Figure 4. The iteration 3 PDM.



7. "Disaster Strikes" and the Requirements Change

At the end of the third iteration our users chose to install the system and start working with it because enough functionality was in place that the system could start earning value early. At that point, they decided to stop development for awhile to see how well the system actually worked. This was good for them because they could start generating revenue with the system, thereby reducing their financial risk by shortening the payback period. It was good for the development team because it provided an opportunity for concrete feedback based on the actual usage of the system in production.

After a few months they realized that they needed to rethink what they wanted (experience does that sometimes). Our stakeholders originally thought that they would like to run tournaments to earn extra money. After talking with a few people they discovered that tournaments involve a lot of work and you're lucky to break even. However, it's still important to run special events such as small internal tournaments and special training sessions where advanced techniques are taught. They also realized that they had forgotten to tell us about the family memberships and children memberships which they offer. Furthermore, they have students studying other styles such as Tai Chi and cardio kickboxing, and even some people studying several styles.

The end result was that we needed to rework our requirements to reflect the new requirements and priorities. These requirements were captured as user stories on index cards, summarized in Table 2. The developers estimated the effort to implement each requirement and the stakeholders prioritized them, enabling us to assign the user stories to coming iterations. The stakeholders are still free to change their minds at any time, introducing new requirements or reworking existing ones, and the developers will continue to work away at the requirements in priority order. Having the requirements change like this helps to illuminate the advantages of an incremental approach to software development. By releasing a portion of the functionality early we enabled our users to better identify what they actually wanted. Had we tried to implement all the requirements at once we would have delivered functionality they didn't actually need in practice.

Lesson #6: trying to define all the requirements up front is a risky proposition.

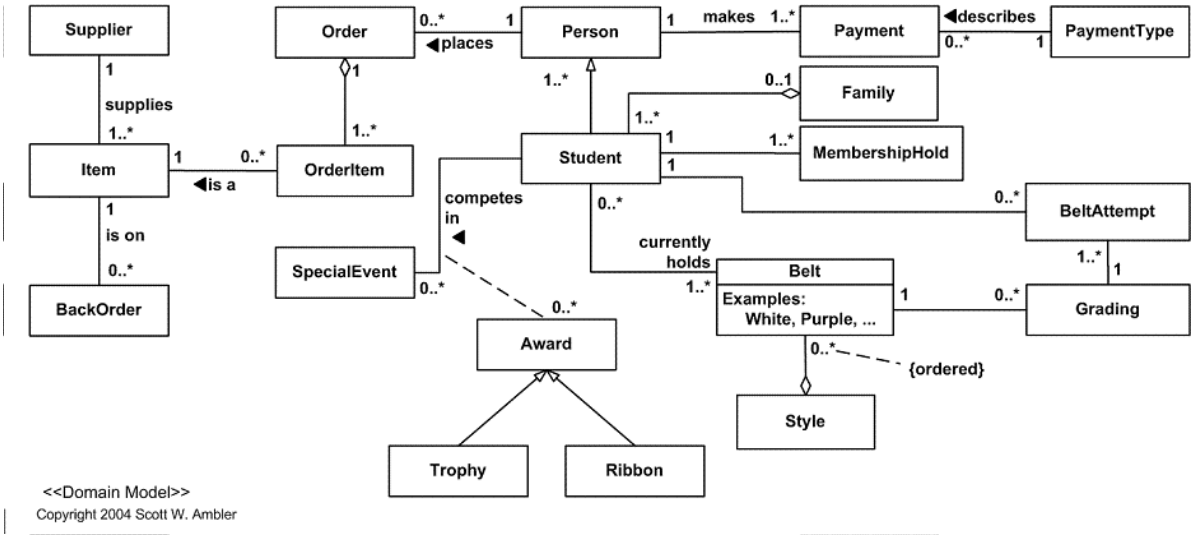
Table 2. Updated User Stories.

Iteration	User Stories
1	• Maintain student contact information • Enroll student • Drop student • Record payment
2	• Promote student to higher belt • Invite student to grading • Email membership to student • Print membership for student
3	• Schedule gradings • Print certificate • Put membership on hold
4	• Enroll child student • Offer family membership plan • Support child belt system
5	• Enroll student in Tai Chi • Support Tai Chi belt system • Enroll student in cardio kick boxing • Support the belt order for each style
6	• Maintain product information • Sell product
7	• Print catalog of products • Order product for inventory • Order product for student
8	• Organize internal special event (special classes, internal tournaments, ") • Enroll student in special event • Print special event certificate for student

8. The Updated Domain Model

The first step is to update the domain model to reflect the changed requirements. As you can see in Figure 5 the changes, compared with the initial domain model of Figure 1, aren't much. In this case (pun intended) I would simply update the whiteboard with other members of the team, involving them with the updates and doing so in an easily visible manner because the model is displayed publicly. The *Tournament* entity has been renamed *SpecialEvent* and is now related to *Student* instead of *Person* because we're no longer including others from outside of the dojo. The *Style* entity has been added to support the ability to offer more than just Karate lessons and the *Family* entity to support family memberships.

Figure 5. Updated Domain Model.



Lesson #7: Shared whiteboard space which is owned by the team can significant enhance communication and productivity.

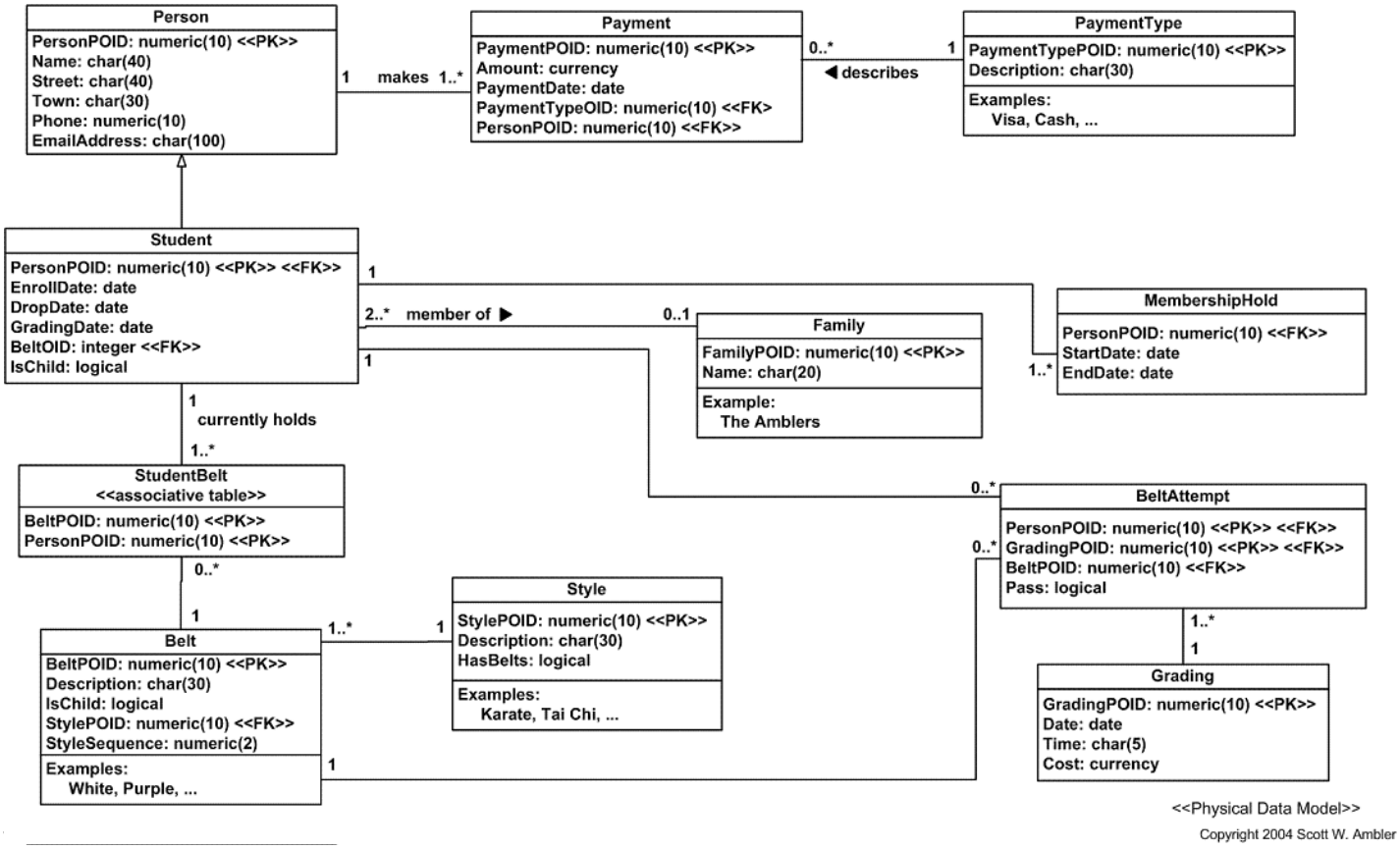
9. Iteration 4

For this iteration we have three user stories to implement: Enroll child student, Offer family membership plan, and Support child belt system. Adding support for child memberships was bit of work. Children have a different set of belts than adults do. Children also have striped belts (white with stripe, yellow with stripe, "  ") in addition to the normal adult colors and two additional colors: red and purple. Kids have more belts in order to keep them engaged. Most adults understand that it could take six to twelve months to earn their next belt but try explaining that to a four year old. In addition to the application code changes we added an *IsChild* column to the *Belt* table as well as new rows for the child belts. We also needed an *IsChild* column to the *Student* table as well. People progress from the children to the adult classes when they've reached an appropriate level of maturity and skill, not just because of their age, so a birthdate column wasn't appropriate. To support family memberships we added the *Family* table to keep track of who was in a given family. We added a corresponding *FamilyPOID* column in *Student* to act as a foreign key to the new table. Most students are not on a family membership so this column will often have a null value.

Figure 6. The iteration 4 PDM.

10. Iteration 5

For this iteration we have five user stories to implement: Enroll student in Tai Chi, Support Tai Chi belt system, Enroll student in cardio kick boxing, Support Tai Chi belt system, and Support the belt order for each style. This iteration focused on supporting non-Karate styles of training, each of which has its own approach to belts. For example Tai Chi only has white belt and black belts and cardio kickboxing doesn't have any belts at all. A *Style* table was added to implement the current requirements and it may make it easy to support new styles in the future although we won't know for sure until we have actual requirements to do so. The *StylePOID* column was added to the *Belt* table to indicate which style a given belt is for - there would be a white belt record for Tai Chi as well as for Karate.



To record the fact that someone can train in several styles we introduced the *StudentBelt* associative table which implements the many-to-many association between *Student* and *Belt*. The Java code, however, does not have a corresponding *StudentBelt* class because Java natively supports many-to-many associations via collections. Data professionals will introduce associative tables to their designs quite naturally, and similarly Java programmers will add collections to their business classes quite naturally, but each group may not be familiar with the techniques of the other group. Luckily Beverley and I were working closely together and were able to map the two schemas effectively once we discovered that there were differences.

> Lesson #8: You can always learn new skills from someone else.
>
> Lesson #9: It isn't enough to specialize in one aspect of technology.

To initialize the *StudentBelt* table we needed to migrate the data from the original *StudentPOID* and *BeltPOID* columns of the iteration four data schema. Each time we rework the existing schema we may need to migrate existing data. This is true for the test data that we use in our development environments and the actual production data. Regular data migration is the downside of evolutionary database development. Migrating data can be difficult, and it's very easy to say that this increased complexity is why we should develop the database schema up front early in the project. Unfortunately this position isn't realistic - if data professionals are going to be relevant on agile development projects then they need to adopt agile development techniques. Even with a traditional approach you're still going to have to migrate data occasionally, new or updated requirements slip in regardless of how well your "change management" process tries to prevent it, so my advice is to accept this fact and get good at data migration. The Process of Database Refactoring article describes how to safely and simply modify your database schema.

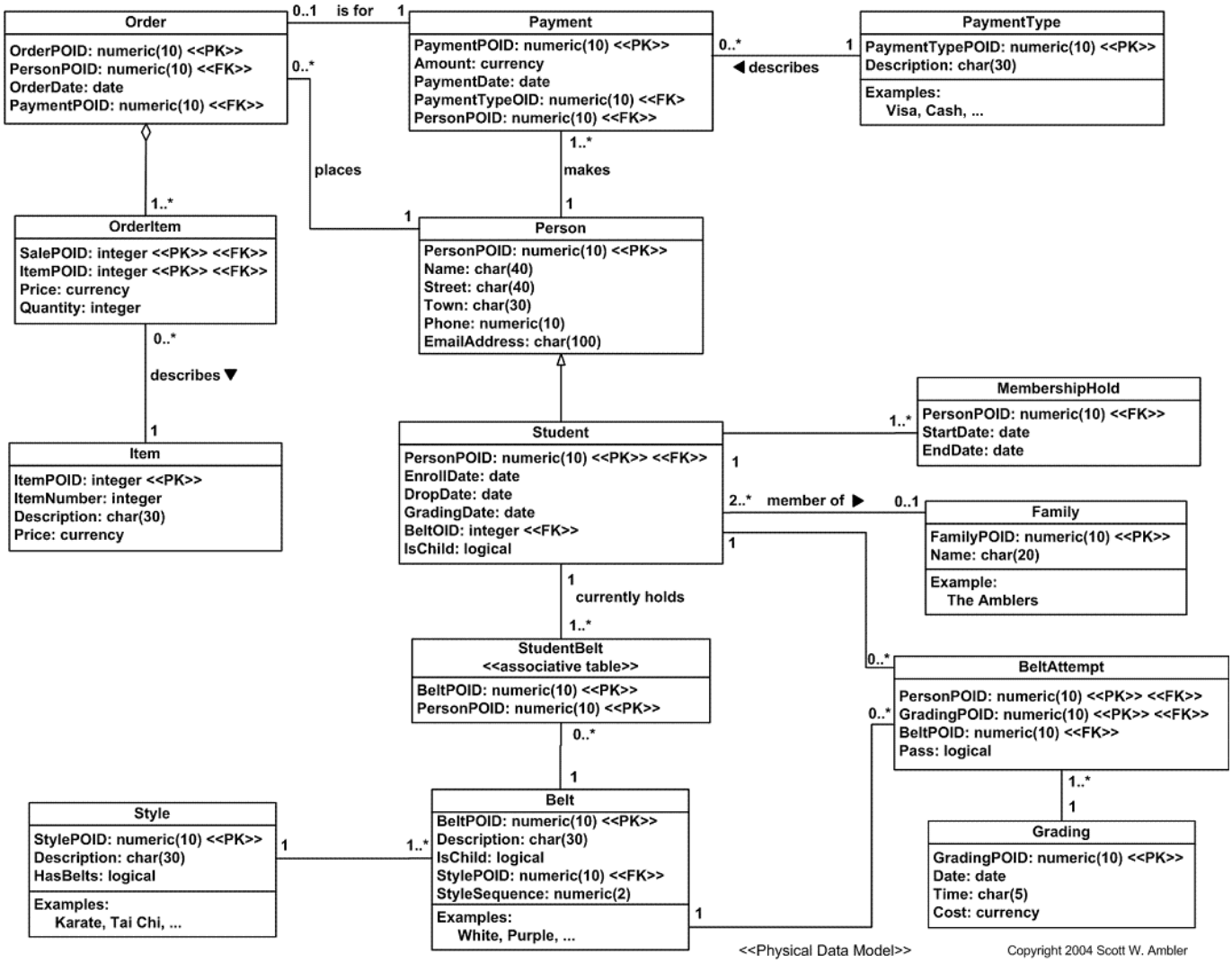
> Lesson #10: Agilists choose to embrace some tasks which traditionalists prefer to avoid.

The final change to the schema was the addition of the *StyleSequence* column to the *Belt* table. We needed to support the fact that people earn belts in a given order: adult Karate students move from white to yellow to orange and so on whereas Tai Chi students move from white to black. Each system has its own order to earning belts.

11. Iteration 6

For this iteration we have two user stories to implement: Maintain product information and Sell product. These additions are very straightforward from a data modeling point of view, we simply added the *Order*, *OrderItem*, and *Item* tables to handle this basic functionality. Notice how these tables are fairly simple for now. For example we're not maintaining stock levels yet nor are we maintaining supplier information. Although we will likely need that sort of information for future requirements we don't need them now so we won't implement them now.

Figure 8. Iteration 6 PDM.



12. Going Straight to Physical Data Modeling

Would it make sense to skip the initial domain model and go straight to physical data modeling? Officially the answer is yes, agile modelers will work in any order that makes sense for their environment and will apply the right artifact(s) as appropriate. In fact, my July 2004, August 2004, and September 2004 columns in Software Development show exactly such an approach for this case study. When you compare the results of the two approaches I believe it is clear that there are several advantages to starting with an initial domain model:

- Both our object schema and data schema could be based on a common model, reducing the chance of a major divergences. Granted, by working together closely and evolving both schemas in parallel this shouldn't happen anyway.
- We could develop a physical schema which reflected future requirements without significant overbuilding.
- We didn't need to invest significant effort in the development of the domain model by focusing just on the fundamental structure and not on details.
- We established common business terminology early in the project, helping us to understand the domain.

However, there were a few trade-offs:

- We overbuilt the schema in the first iteration by introducing the *Person* table before we needed it. This resulted in a slight reduction in performance due to the need to join *Person* and *Student* to persist student information.
- We did have to invest some time to create the initial model.
- Although the domain model was fairly basic to our stakeholders it was still far more abstract than working software. We risked them thinking that we were wasting their time with unnecessary artifacts (granted, you should always be prepared to explain why you're doing whatever it is that you're doing).

13. Critical Lessons in Agile Data Modeling

Throughout this article I identified a collection of lessons which I believe are critical to your success at agile data modeling. These lessons are:

1. **Agile data modelers travel light and create agile models which are just barely good enough**. Agile data models are just barely good enough. Agilists solve today's problem today and trust that they can solve tomorrow's problem tomorrow.
2. **Agile data modeling is both evolutionary and collaborative**.
3. **You can be agile yet still support the needs of the enterprise**. You can think about the future, and act on it, in a very agile manner if you choose to.
4. **Agile data models can and should follow your enterprise standards**. In fact, following modeling standards is an AM practice. Your standards should be straightforward, simple, and sufficiently described so that the team can learn and then follow them. Critical data modeling standards focus on naming conventions and conventions for writing stored procedures.
5. **Trying to define all the requirements up front is a risky proposition**. Requirements change over time, so embrace this concept and adopt techniques which allow you to react effectively. In short, avoid big requirements up front (BRUF).
6. **Shared whiteboard space which is owned by the team can significantly enhance communication and productivity**. Sketching is the most common primary approach to

modeling.

7. **You can always learn new skills from someone else**. This is one of the many benefits of collaborative development.
8. **Don't be just a data modeler**. Become a generalizing specialist with a wider range of skills.
9. **Embrace "hard" tasks**. Many traditionalists think that data migration is hard, and it is, but if you choose to get good at it you'll soon discover that it's not so bad after all. Data migration is an important part of implement most structural database refactorings, and database refactoring is a critical activity which supports evolutionary database development, so you'd better get good at it.

Repeat after me: comprehensive data models are not required up front, comprehensive data models are not required up front, comprehensive data models are not required up front, ...

14. Agile Data Modeling in Context

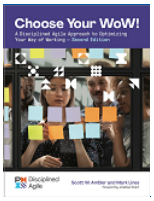
The following table summarizes the trade-offs associated with agile data modeling and provides advice for when (not) to adopt it.

Advantages	• Enables evolutionary exploration of both your problem and solution spaces • Enables decisions at the most responsible moment • Integrates into other agile ways of working
Disadvantages	• Traditional data modelers struggle with agile data modeling strategies at first, particularly when they don't yet have full stack agile database skills. • Requires ability to safely refactor whatever is being modeled.
When to Adopt This Practice	• Modeling to think things through before you implement something is always advised in all but the most trivial of situations. • Data modeling is always advised when you evolving data sources, and agile data modeling advised except when you are explicitly taking a traditional/serial approach to development (even then, some aspects of agile data modeling such as working collaboratively should still be considered).

15. Related Resources

- Agile Data Architecture
- Clean Database Design
- Database Refactoring
- The Database Techniques Stack

Recommended Reading



This book, Choose Your WoW! A Disciplined Agile Approach to Optimizing Your Way of Working (WoW) ♦ Second Edition, is an indispensable guide for agile coaches and practitioners. It overviews key aspects of the Disciplined Agile♦ (DA♦) tool kit. Hundreds of organizations around the world have already benefited from DA, which is the only comprehensive tool kit available for guidance on building high-performance agile teams and optimizing your WoW. As a hybrid of the leading agile, lean, and traditional approaches, DA provides hundreds of strategies to help you make better decisions within your agile teams, balancing self-organization with the realities and constraints of your unique enterprise context.

I also maintain an agile database books page which overviews many books you will find interesting.

Disciplined Agile | Agile Modeling | Agile Data | Ambysoft | @scottwambler

---