

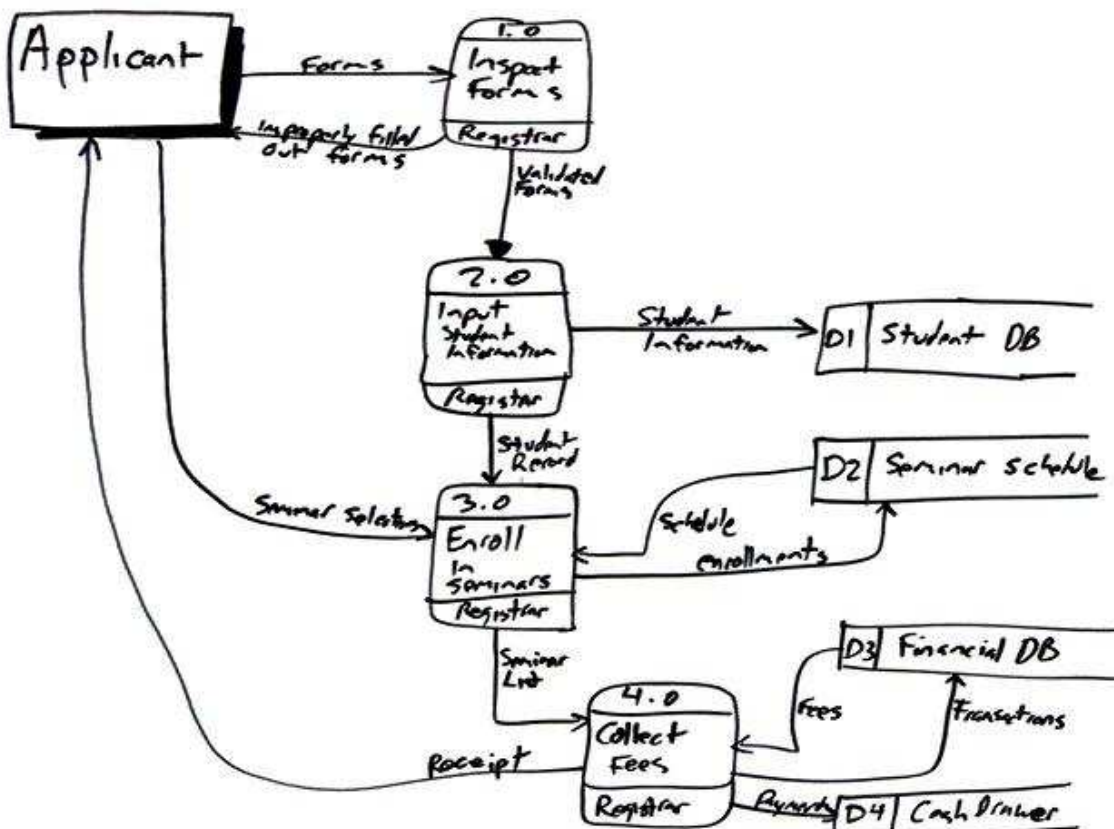
In the late 1970s *data-flow diagrams (DFDs)* were introduced and popularized for structured analysis and design (Gane and Sarson 1979). DFDs show the flow of data from external entities into the system, showed how the data moved from one process to another, as well as its logical storage. **Figure 1** presents an example of a DFD using the Gane and Sarson notation. There are only four symbols:

[Ads by Google](#)

1. Squares representing *external entities*, which are sources or destinations of data.
2. Rounded rectangles representing *processes*, which take data as input, do something to it, and output it.
3. Arrows representing the *data flows*, which can either be electronic data or physical items.
4. Open-ended rectangles representing *data stores*, including electronic stores such as databases or XML files and physical stores such as filing cabinets or stacks of paper.



Figure 1. Enrolling in the university.



To create the diagram I simply worked through a usage scenario, in this case the use case logic described in the **Enroll in University** system use case. On actual projects it's far more common just to stand at a whiteboard with one or more project stakeholders and simply sketch as we talk through a problem.

In this case I started with the applicant, the external entity in the top left corner, and simply followed the flow of data throughout the system. I introduced the *Inspect Forms* process to encapsulate the initial validation steps. I assigned this process identifier 1.0, indicating that it's the first process one the top level diagram. A common technique with DFDs is to create detailed diagrams for each process to depict more granular levels of processing. Were I to do this for this process I would number the subprocesses 1.1, 1.2, and so on. Subprocesses of 1.1 would be numbered 1.1.1,

1.1.2, and so on. I wouldn't bother to expand this process to more detailed DFD as it is fairly clear what is happening in it and therefore the new diagram wouldn't add any value. I also indicated who/what does the work in the bottom section of the process bubble, in this case the registrar. This information is optional although very useful in my experience. You can see how the improperly filled out forms are returned to the applicant if required.

I then continued to follow the logic of the use case, concentrating on how the data is processed by each step. The second process encapsulates the logic for creating a student record, including the act of checking to see if the person is eligible to enroll as well as if they're already in the database. Notice how each data flow on the diagram has been labeled. Also notice that the names of the data change to reflect how it's been processed.

Now that I look closer at the diagram the arrow between the *Input Student Information* process and the *Student DB* data store should be two-way because this process searches the database for existing student records. Unfortunately I've erased this diagram from my whiteboard so it isn't easy to address this minor problem. Yes, I could use a drawing program to update the arrowhead but it's more important to make the point that agile models don't need to be perfect, they just need to be good enough. AM recommends that you follow the practice **Update Models Only When it Hurts** and in this case this issue doesn't hurt enough to invest the two or three minutes it would take to fix the diagram.

The *Collect Fees* process is interesting because it interacts with an electronic data store, *Financial DB*, as well as a physical one, *Cash Drawer*. DFDs can be used to model processes that are purely physical, purely electronic, or more commonly a mix of both. Electronic data stores can be modeled via data models, particularly if they represent a relational database. Physical data stores are typically self explanatory.

There are several common modeling rules that I follow when creating DFDs:

1. All processes must have at least one data flow in and one data flow out.
2. All processes should modify the incoming data, producing new forms of outgoing data.
3. Each data store must be involved with at least one data flow.
4. Each external entity must be involved with at least one data flow.
5. A data flow must be attached to at least one process.

Although many traditional methods have a tendency to apply DFDs in dysfunctional ways it is still possible to do so in an agile manner as well. Keep your diagrams small, as I did above. Use simple tools, such as whiteboards, to create them with your stakeholders. Travel light and erase them when you're through with them. Create them if they're going to add value, not simply because your process tells you to do so. The bottom line is that some of the modeling methodologies may have been flawed but the need to represent the data flow within a system is still required.

Source

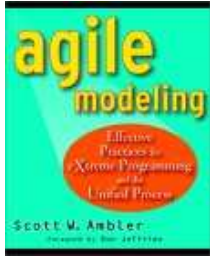
This artifact description is excerpted from Chapter 9 of **The Object Primer 3rd Edition: Agile Model Driven Development with UML 2**.

Suggested Reading

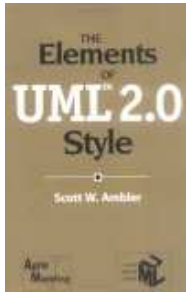
- **Artifacts for Agile Modeling: The UML and Beyond**
- **Modeling Style Guidelines**
- **Security Threat Modeling**
- **Why Extend the UML Beyond Object and Component Technology?**



The Object Primer 3rd Edition: Agile Model Driven Development with UML 2 is an important reference book for agile modelers, describing how to develop 35 **types of agile models** including all 13 **UML 2 diagrams**. Furthermore, this book describes the techniques of the **Full Lifecycle Object Oriented Testing (FLOOT)** methodology to give you the fundamental testing skills which you require to succeed at agile software development. The book also shows how to move from your agile models to source code (**Java** examples are provided) as well as how to succeed at implementation techniques such as **refactoring** and **test-driven development (TDD)**. The Object Primer also includes a chapter overviewing the critical database development techniques (**database refactoring**, **object/relational mapping**, **legacy analysis**, and database access coding) from my award-winning **Agile Database Techniques** book.



Agile Modeling: Effective Practices for Extreme Programming and the Unified Process is the seminal book describing how agile software developers approach **modeling** and **documentation**. It describes principles and practices which you can tailor into your existing software process, such as **XP**, the **Rational Unified Process (RUP)**, or the **Agile Unified Process (AUP)**, to streamline your modeling and documentation efforts. Modeling and documentation are important aspects of any software project, including agile projects, and this book describes in detail how to **elicit requirements**, **architect**, and then **design** your system in an agile manner.



The Elements of UML 2.0 Style describes a collection of standards, conventions, and **guidelines** for creating effective **UML diagrams**. They are based on sound, proven software engineering principles that lead to diagrams that are easier to understand and work with. These conventions exist as a collection of simple, concise guidelines that if applied consistently, represent an important first step in increasing your productivity as a modeler. This book is oriented towards intermediate to advanced UML modelers, although there are numerous examples throughout the book it would not be a good way to learn the UML (instead, consider **The Object Primer**). The book is a brief 188 pages long and is conveniently pocket-sized so it's easy to carry around.

Translations

- **Japanese**

Let Me Help

I actively work with clients around the world to improve their information technology (IT) practices as both a mentor/coach and trainer. A full description of what I do, and how to contact me, can be **found here**.



Copyright © 2003-2006 **Scott W. Ambler**

Last updated: April 3, 2006

This site owned by **Ambysoft Inc.**

|| **Agile Data (AD)** | **Agile Unified Process (AUP)** | **Enterprise Unified Process (EUP)** | **My Writings** ||