



Written By Eamon O'Tuathail

Introducing a Matrix Of Models

Overview

A typical developer encounters the word “model” a dozen or more times in a day in various contexts. A model is a simplified yet accurate representation of a system (the code) that helps in its understanding from a particular perspective. I like to think of a model as a context for making decisions surrounded by all relevant facts and just enough other information so as to help us be productive and yet not totally swamp us. Developing and deploying software can be regarded as involving a series of overlapping models, even if many software teams do not consider it as such.

Models have got a bad name for a number of reasons we have all seen. Incomprehensible diagrams with boxes and arrows covering entire walls of conference rooms, telephone-book size “architecture” documents or months and months of “analysis” before any real programming begins are characteristics of all too many software projects that usually result in little satisfaction among their participants – those who write the software ... those who use the software ... and those who pay the bills.

Yet as software developers we use models all the time and they clearly bring numerous benefits. Though a few developers are happy dealing with a million lines of code in one go - the rest of us mere mortals are more productive with simplifications that allow us to focus better. Often people other than software developers have a strong interest (and can make useful contributions) in how software evolves. The real-world usage of the software needs to be understood during software development so that the results are better aligned with needs.

A Matrix Of Models

There are many good books that focus on a single model (one might cover the domain model, another the entity data model and yet another the security model), but our goal here is to review the rich variety of software models that are available (there are more than you initially think) and to explore how we can make them all work together productively. Sometimes the models are layered (think of user interaction model talking to domain model taking to entity data model), but more often it is better to think of them as a matrix. Each model influences other models and delegates responsibilities to and accepts obligations

from them - e.g. think of the relationship between the deployment model / cloud (hosting) model / behavior (requirements) model.

Model Categories

You should use as many models as makes sense for your particular project. Some models are architectural and some are operational in nature. Some teams will limit themselves to three or four while others will use lots (e.g. 10+) . The number is not important - what is is that they when taken together they provide a holistic view of the system that cater to the interests of all important stakeholders.

Larger projects tend to have more models and each model should play a significant part in the project. It is quite possible a team may need to create its own model types in addition to (or instead of) the models we discuss here. Smaller projects may well merge the concerns covered by multiple models into one. Here we discuss User Interaction Model (user interface) separate from the Learning Model (user education/documentation) but some smaller projects may treat them as one.

We will divide potential models into six categories - core models, developer-focused models, knowledge representation models, code models, stakeholder-focused models and business-focused models. Let's first look at each category in turn.

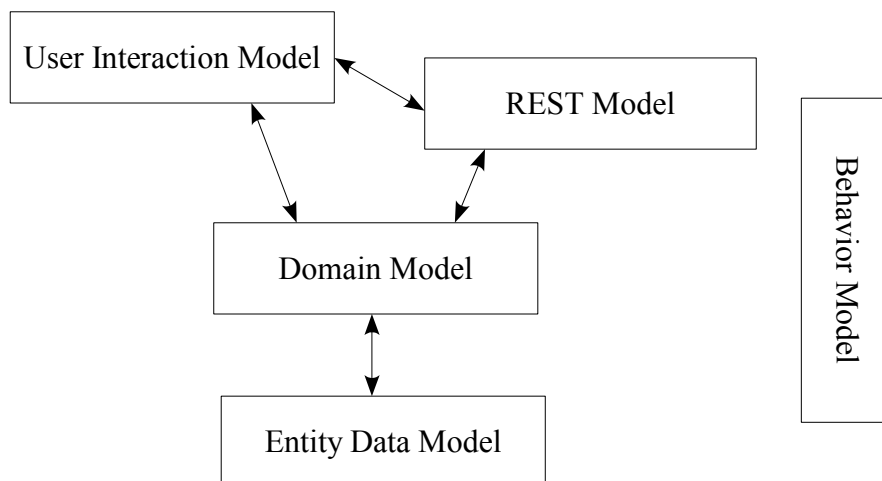
Core Models

The core models provide the foundation for software development:

- Behavior Model (also called requirements, user stories) - what we want to achieve, from users' perspective
- Domain Model - guides the definition of core concepts within the problem space; includes relationships and rules
- Entity Data Model - guides how data is persisted, queried and transformed
- User Interaction Model - guides how users experience the solution
- REST Model - guides how state is transferred over distributed networks, such as web services or data syndication

So, how is that different from your traditional n-tier enterprise app with a user interface layer, distributed messaging layer, business logic layer and data access layer? It is and it isn't. As we said at the outset, developers are using models all the time. Much of what we discuss here is already well known in the software community, yet I wish to bring certain characteristics of how we (should) develop software into sharper focus.

Core Models



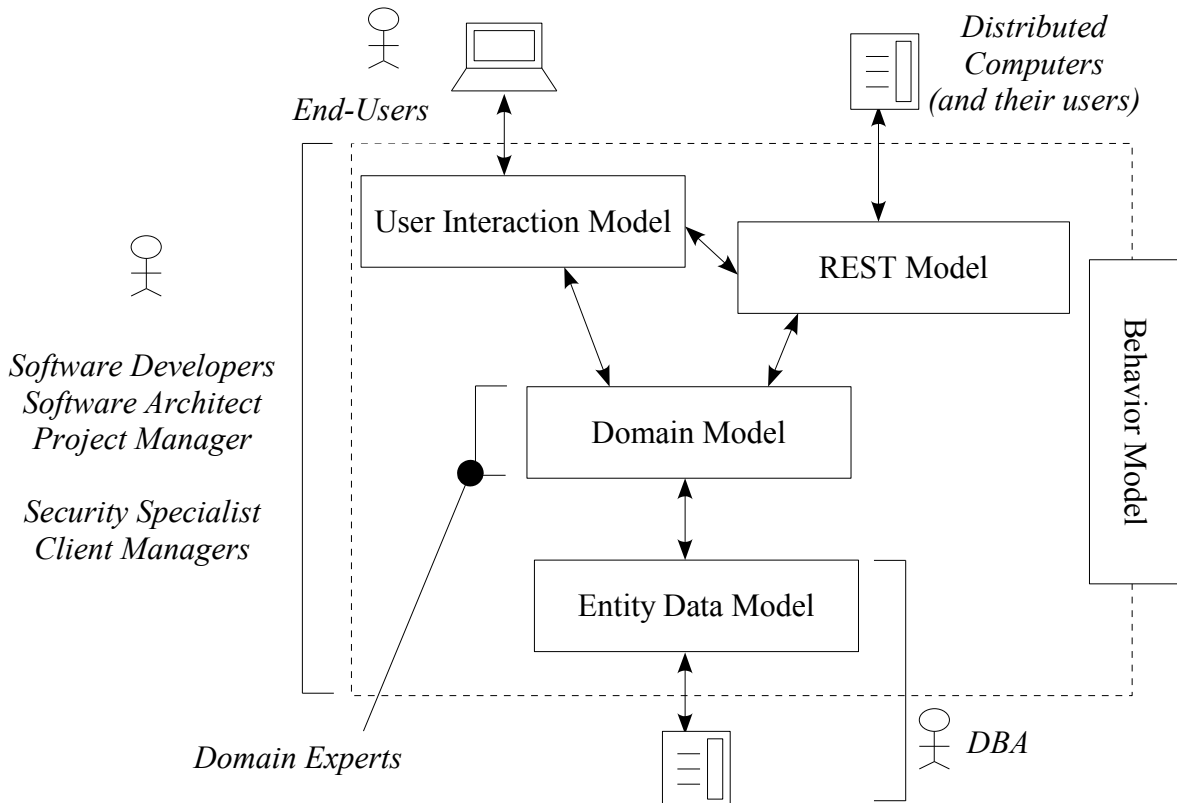
A key point is that these models do not necessarily correspond to software layers. Some simple projects will put all code for the User Interaction Model into a single user interface DLL, but many modern apps have more demands - user interaction in a Web 2.0 scenario will make calls to the REST Model to exchange state, styling in the form of CSS3 will be used, part of the user interface will run in the web browser (HTML5, Javascript, Web Workers, GeoLocation) and issues such as globalization and accessibility need to be considered. The User Interaction Model needs to consider many issues, and it is not just a simple UI page layout that is involved. The same applies to other models.

A software layer may involve one or more components. It is quite possible that a single model will require code in multiple layers of software, or that a single layer (or a single DLL) will contain code for different models. For example, when using ASP.NET MVC, the server-side code for user interaction is there, the client-side code is elsewhere and the `IController`-derived controller classes could also deliver the REST model (alternate designs could include using the RESTful features of WCF Web API, or ADO.NET Data Services and these could be in separate layers). So models are separate from layers. The idea of a single layer deliver multiple models will become more prevalent when we soon look at other models. For now, consider the security model and how there is no “security layer” - Where do you think security should be implemented? Everywhere, of course!

A software model may transcend the network. The User Interaction Model appears on the client and server side of the HTTP connection. The Entity Data Model is influenced both by the conceptual model on the application and the data objects stored in the database itself.

Many people and external systems interact with these core models. So we need to add to our picture the development team, end-users, remote systems and the database engine and its DBA.

Core Models And Their Stakeholders



What do models bring to the table? Quite a bit. Models help to more clearly delineate responsibilities. A coordinated set of models can intersect to deliver robust solutions primed for ongoing evolution. Yet when needed, by focusing on a single model at a time, and what it represents and the concerns for which it is responsible for addressing - development teams can gain a deep understanding of the optimum way of delivering what is needed, avoid re-work and overlap and missed needs.

Developer-Focused Models

Developer-focused models include:

- **Development Model** - how a software team goes about developing software; guides the team's organization, work allocation and decision making regarding project iterations, build integrations, etc.
- **Deployment Model (installer)** - how to get the executable bits/datastore

onto the host machine(s) and how to upgrade them from time to time

- Test Model - How we verify we are delivering what's needed
- Technology Model - which software (APIs, datastore, tools, programming languages) we use to build the platform

Knowledge Representation Models

Knowledge representation models include:

- Semantic Model (OWL/RDF representation) - subject/predicate/object statements about resources we work with
- Event Model (logging, tracing, events, instrumentation, undo/redo, temporal history) - how to track what is going on
- Analytical Model (data mining, multidimensional data) - analysis of data
- Business Intelligence (BI) Model (Reporting) - how to extract actionable results from data

Code Models

Code models include:

- Identity Model - What identity-related information we need to manage and how we ensure we protect what needs to be protected
- Extensibility Model (API/Framework) - How third parties can extend the platform in an orderly manner
- Infrastructure Model - the utility functionality (other than data) that exists below the domain model
- Integration Model - How external systems are integrated with the platform

Stakeholder-Focused Models

Stakeholder-focused models include:

- Maintenance Model - When bugs are detected, what happens to ensure delivery of a fix
- Admin Model - how system administrators manage the user population and admin-level functionality
- Hosting Model (Operations) - deciding how the platform is brought to life on hardware
- Learning Model (User Education/Documentation) - helping the userbase

learn how to use the platform

- Product Feature Model – Organizing product lines as a collection of overlapping products into features, components, editions and versions and managing their evolution

Business-Focused Models

Business-focused models include:

- Business Model – everything surrounding business aspects of software creation
- Sales Model – Helping customers purchase the software
- Partner Model – Deciding how other companies can work with us to deliver value to customers
- Professional Services Model (consultancy/custom development) – consultancy and other forms of people-delivered assistance to help client companies in the user of the software platform
- Community Model – Deciding how users (possible from different companies) can help each other (forums, meetups, articles, etc.)
- Support Model (technical support/helpdesk) – How user queries are handled efficiently

Review

There you have it, more than two dozen potential models. An experienced developer will have heard of many of these models, and a few may be new. The naming of some may be slightly different, but that is not too important. For smaller projects it is certainly likely you will not use all of these. For some projects you may get by with merging models, which is fine. For your own projects you may need to add more model types.

Representing Models

A software solution consists of a cohesive set of models. Running code is the canonical representation of all core and developer-focused models. Stakeholder- and business-focused models are represented by arrangements among people, business contracts and work patterns. For the Developer models, we need to examine how to represent them.

There are multiple developer-focused models (e.g. domain model, user interaction model, security model, deployment model) and though they look at different issues, they are not meant to conflict. It is running code that is the “Single Truth” and keeps all the models integrated and consistent. However,

code on its own is not enough.

For example, most enterprises have a team of security experts - usually from a software engineering background, with additional security training (e.g. CSSIP-Certified Information Systems Security Professional) and a good understanding of the enterprise's overall security landscape. Before any software solution, regardless of whether it is internal or external, is installed in the enterprise's data center, the skeptical security team must be convinced that it will not jeopardize the security of their network. In essence, they want to be presented with the security model for the solution. This is not the million lines of C# or JAVA source code that interests them, rather a simpler yet still accurate representation that explains to them how the solution's security infrastructure works. This is usually a document with text and graphics, but could also be aided by a face-to-face presentation combined with a question & answer session. The security model must not be completely divorced from the codebase (how to keep models and the codebase aligned will be discussed in some detail later) - instead it must be an accurate representation of the codebase from the viewpoint of security experts. It should include all information pertinent to security and how it relates to the rest of the software solution.

Even within the development team, models can be useful. Put a few software developers in a meeting room and within ten minutes one of them will be up at the whiteboard drawing. It is simply more efficient if what they draw is from a commonly understood graphical language - and UML is what gets used, it is widely understood and it is comprehensive (well, too comprehensive!). So sketching in UML is one good way of representing models (shock, horror, we said something nice about UML).

Where the UML über-enthusiasts have gone wrong is to try and recreate in their diagrams everything that is represented in code and to the same level of detail. Similarly, those software teams who write verbose design documents containing thousands of pages of text are making the same mistake. No one reads them, developers hate to write them and they are never updated in lockstep with the codebase (regardless of diktats from senior management). They are a wasteful drag on the development team's productivity.

Models can be represented by transient or permanent artifacts. Models can be manually created or auto-generated from the codebase or a mix. We have a strong preference for auto-generated artifacts but realize some have to be manually written. We wish to minimize the number of permanent artifacts that need to be manually maintained. Models evolve over time so we need to be aware when they get out of sync.

Any permanent artifacts consume precious time to keep up to date so we wish to

ensure they deliver rewards in productivity gains for developers, additional features in the product or in some other way bring real value to customers. What this means in practice depends on the model. For example, I like to treat the documentation for the Domain Model as user-oriented documentation (roughly equivalent to what Eric Evans in his wonderful book would call exploratory domain models). The Domain Model is meant to be in language from the users' world (ubiquitous language) so the description of this model should be understandable and verifiable by them (if fact, it is critical that they do ensure it conforms to their world). A domain model also requires implementation in code, and for this smaller (or perhaps no) additional developer-focused documentation is needed. Another example is using user documentation as an excellent description of the User Interaction Model.

An an example of auto-generated artifacts consider how the Entity Data Model could be described by the data dictionary and data diagrams that development tools/databases can produce from information they store directly.

You have been using these models everyday during software development, just not considering them as an integrated set, which of course they are (or at least, should be).

Architecture / Product Management / Business Development Binds Models Together

There are lots of models and a concern is how to get work done on them all. You will not do it all at once. For large platforms, there are too many for one person to master and specialization is required.

I like to break them into three areas:

- Architecture (led by architect/project manager)
- Product Management (led by product manager)
- Business Development (led by business development manager or equivalent)

The architecture is how all developer-focused models are integrated. Each of these models will be made available via various artifacts but in addition a thin architectural document is also required. Product management is all the work that turns a software project into a product. Business development is what turns a product into a successful enterprise (whether the software goes to internal customers or external).

On smaller projects, all three areas may be led by the same person. On larger projects, each may involve multiple people.