astah

Modeling in the Agile Age:

What to Keep Next to Code to Scale Agile Teams

Kenji Hiranabe, Change Vision, Inc

Now that Agile methods have become mainstream in software development, working code and automated tests are being considered as the most important team artifacts. Is modeling obsolete? Is UML dead?
I don't think so. In this article, I'll explore the spaces where modeling fits and plays an important role in this Agile age, especially when development scales to multiple teams and a shared understanding of the system's "Big Picture" becomes essential.

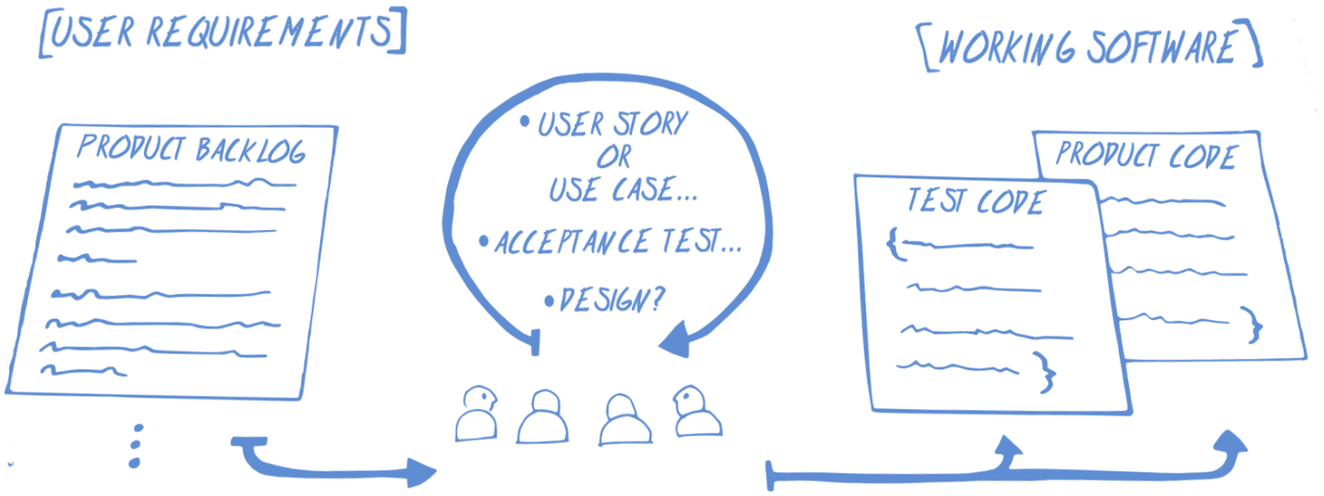


Figure. 1 Simple Scrum framework

Where is "Design" in Agile

"While code is the truth, it is not the whole truth."
-Grady Booch

To begin with, let's describe a minimal process of an Agile team using Scrum. Figure 1 shows an intentionally simplified process with the only essential artifacts left explicit.

- The "User Requirements" are listed as a "Product Backlog".
- The development team picks items from the list and implements them within a short iteration (or "Sprint").
- After each Sprint, the team produces "Working Software" (or "Increments") as "Product Code" and "Test Code".

In this minimum framework, the input to the team is “User Requirements” as “Product Backlog” and the output is “Working Software” as code (“Product Code” and “Test Code”). No other design artifacts between them are explicitly described here. All the design intentions generated during the Sprint will hopefully have been deployed into the working codebase as assets of the team, but there will be information which cannot be expressed directly as code. Scrum is a process-framework and intentionally mentions nothing about design, but we still have design and design activities in our team.

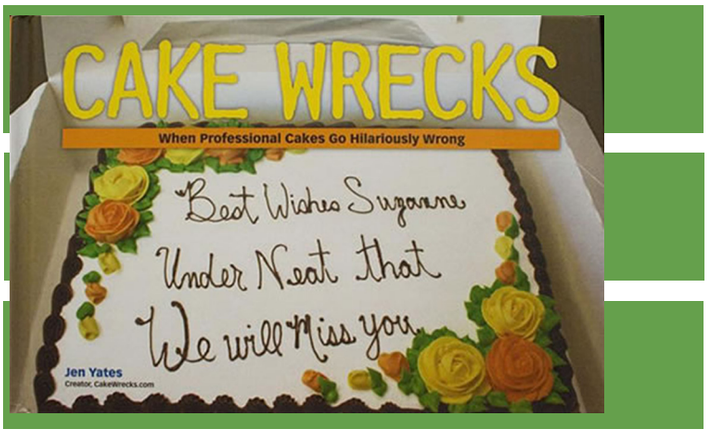
As Grady Booch says, “The code is the truth, but it is not the whole truth.” [Booch99] So if there is information which cannot be expressed or communicated by the code, where can we keep these information assets? That is the question this article attempts to answer.

Documentation is not Agile?

“Model to have a conversation.”
–Craig Larman and Bas Vodde

“In our minds!” would be one answer to the previous question. Daily meetings, pair programming, design workshops, and other social practices act as synchronization and continuous integration of the minds of the team members. But when the team gets bigger, geographically distributed, and people leave the team, “models in minds” will quickly evaporate. We need to maintain the shared understanding of the system somewhere as documentation to share information which is not well communicated and retained only by the code.

One point that Agile makes clear is the value shift from documentation to conversation, so writing heavy design documentation (which often duplicates information from the code) is not the right approach. Documentation that makes conversations effective is the approach that we should take, and it should be the simplest possible set of models which works complementary to the code.



One aspect where models have an advantage over code is visual expression. In other words, text is a poor medium of communication in a certain context. The image above[1] shows how textual communication can fail miserably.

This “wreck” was likely caused by a message left on the answering machine of the cake shop. If the caller had used a simple picture with text, he or she could have easily avoided the wreck. Sometimes, “a picture is worth 1,000 words.”

So what models can be effectively used, and for what purpose in Agile teams?

Agile Modeling and Two Categories of Models

“Let’s keep the modeling baby but throw out the bureaucracy bathwater”

– Scott Ambler

“Agile Modeling” is a set of practices you can use within your Agile teams for effective modeling and documentation [Ambler05]. This method aligns with Agile values and principles and still helps you benefit from the power of modeling. The emphasis is on models for conversation, not for handovers.

We have been using the practices and principles in Agile Modeling with our software development team and found that the most important role of models is to **visually communicate the "Big Picture"** or the "Bird's-eye View" of the system design, which is difficult to accomplish via code. Without it, the team would be "four blind men touching the elephant"[2]. Each man feels only the part he is touching, and it takes a long time to unify the parts into a meaningful whole - the elephant.

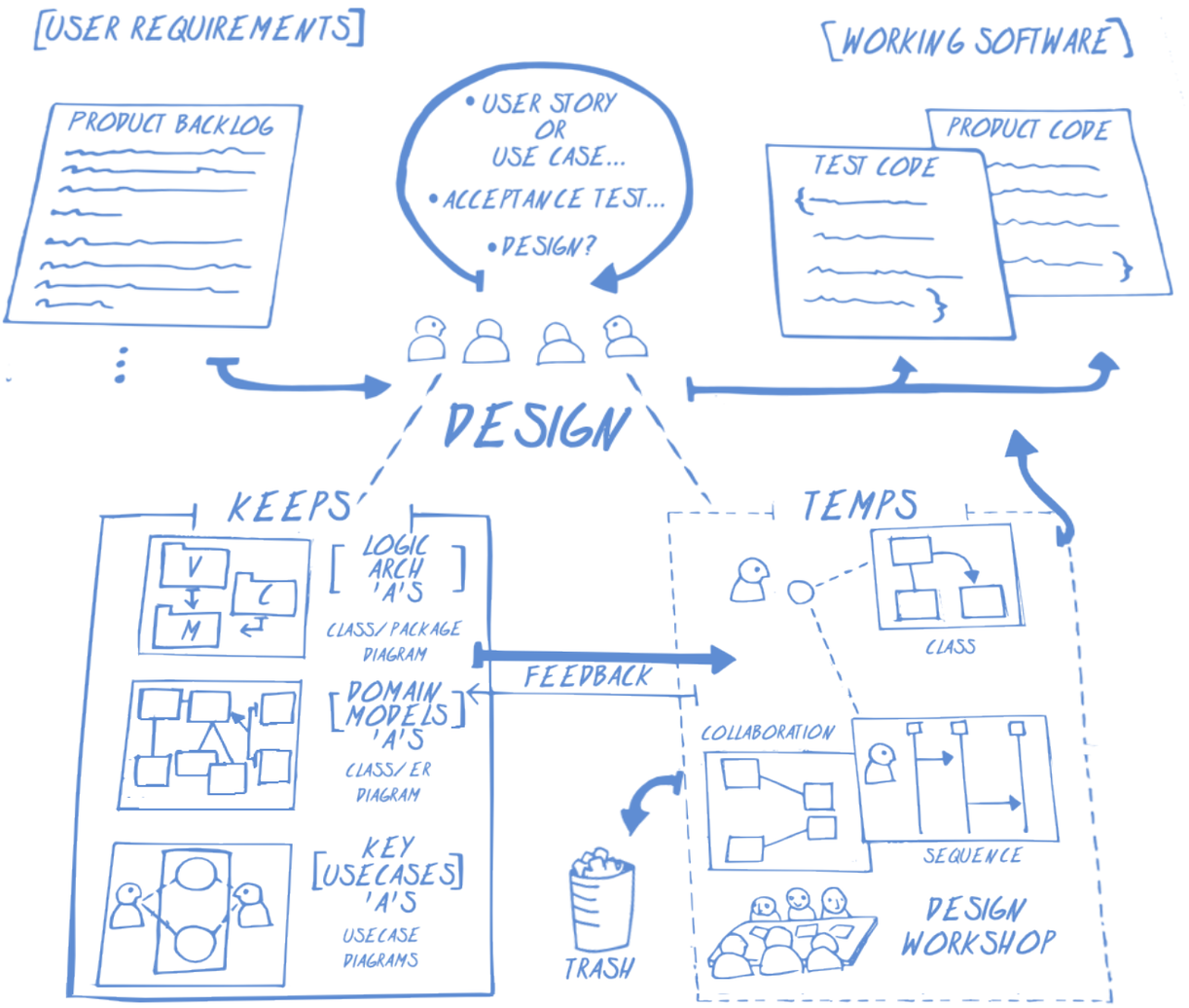
My recommendation of the "Big Picture" models to keep maintained consists of:

1. **"Architecture"** of the system for the team to get a rough idea of the whole system structure.

2. **"Domain Model"** to help the team understand the concepts used in the problem domain.

3. **"Key Use Cases"**[3] to help the team understand the concepts used in the problem domain.

Figure 2 Agile Modeling with "Keeps" and "Temps"



1. Thanks to Jeff Patton for this funny example. See http://www.cakewrecks.com/

2. From "The Blind Men and the Elephant," by John Godfrey Saxe. The first man said "it is a wall", the second "snake", the third "tree", and the forth "rope. [Evans 05]."

3. User Story is used more widely in Agile, but I prefer to use Use Cases in larger development.

These are all essential to establish understanding of the system as a whole. Without models, how would you accomplish this understanding? If you have a large codebase, and make a “Big Picture” assumption based on a small incomplete view of it, you will make some poor choices on how to maintain that codebase. The “Big Picture” not only comprises the team’s mental model of the system but also supports the vocabulary they use in the conversation and in the code that they program, i.e. the structures of the code as well as detailed naming of the programming constructs such as classes, methods, variables, fields, data, and configuration. In other words, these models are not only important for the team to establish a shared understanding of the system as a whole, but also for the team to keep the codebase consistent and maintainable.

On the other hand, there are also temporary models which will be thrown away once the information is programmed into the code. Casual class diagrams with few classes and sequence diagrams describing the interaction of them (usually drawn on whiteboards) fall into this categories. Those models are also important to encourage conversation and to burn the information to the codebase before the models are thrown away. So the core of the idea is to categorize the models into two types -- **models to keep and maintain** as assets and **models to draw temporarily** to have effective conversation. We call the former “keep models” or “Keeps”, and the latter ”temp models” or “Temps,” as described by the illustration in figure 2 on page 3. Please note that the “Keeps” are not meant to be “frozen”, but are meant to be maintained and changed over time. In the next section, I’ll propose three “Keeps” for Agile teams.

Models to Keep

Depending on the context (the number of people, criticality of the system, stability of requirements, whether it is an enterprise system or embedded), the “Keeps” models will vary. But here are good candidates for “Keeps”[4] from my experience.

1. **Architecture As Class/Package Diagrams**
2. **Domain Model As Class Diagram/ER Diagrams**
3. **Key Use Cases As Use Case Diagrams + Sequence/Communication Diagrams**

We mainly use UML, but you don’t have to stick to strict UML specifications. We use it because it includes sufficient standardized diagrams and there are many published educational materials. ERD (Entity-Relation Diagram) and DFD (Data Flow Diagram) for data and processes are also used sometimes for the same reasons.

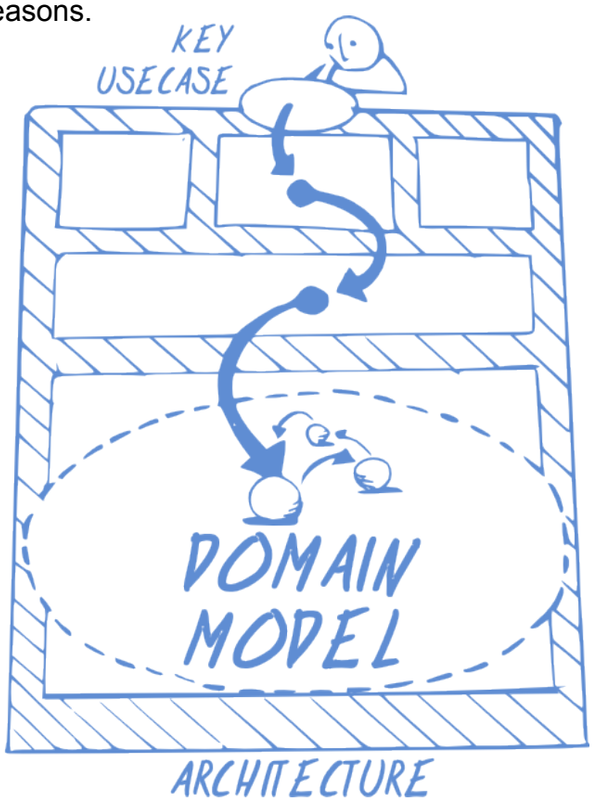


Figure 3 Architecture, Domain Model and Key Use Cases

4. We use mainly UML because it includes standardized diagrams and there are a lot of educational materials published. ERD (Entity-Relation Diagram) and DFD (Data Flow Diagram) for data and processes are also used sometimes for the same reasons.

On the last page, Figure 3 illustrates the roles of the three “Keeps” models as a picture. In a nutshell, “Architecture” shows the structure, “Domain Model” shows the core concepts of the problem space, and “Key Use Cases” shows examples of the usage of the system.

Here are the three “Keeps” models using concrete examples.

1. Architectures As Class/Package Diagrams

The architecture is a structural presentation of the whole system. It is often described by class or package diagrams typically to show global tiers (layers). For example, in an application with UI and database, tiers are usually set horizontally from UI to database, and one Use Case walks through them to accomplish its goal.

Other architecture patterns like “MVC” (Model-View-Controller) can also be chosen as a global architecture. Figure 4 is an example of an architecture drawn as a package diagram

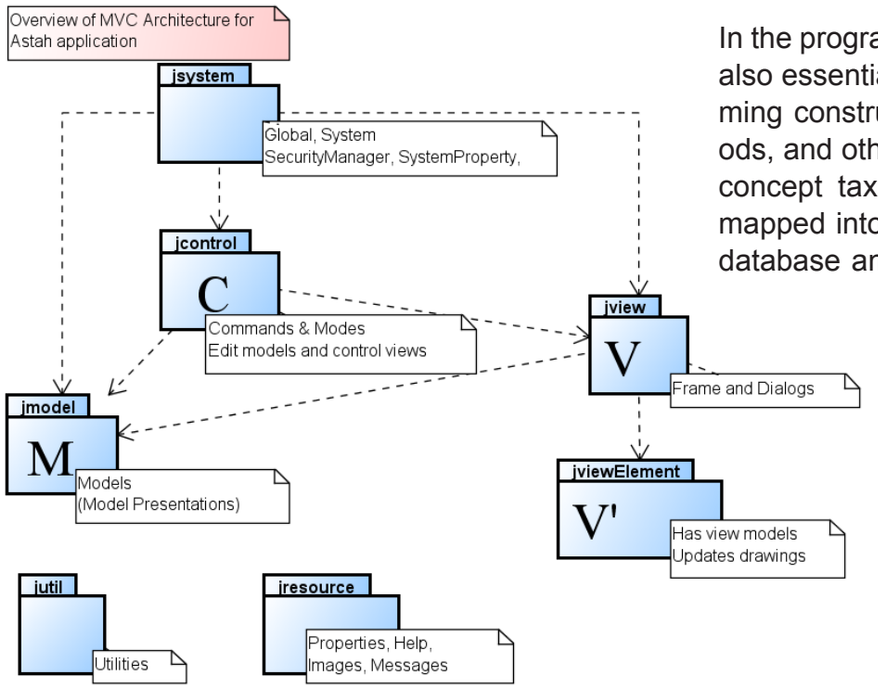


Figure 4 Architecture As Class/Package Diagram

based on MVC architecture.

Everyone in the team should understand the roles and meanings of the components of the architecture so that team members can write code which fits in the right place in the architecture consistently.

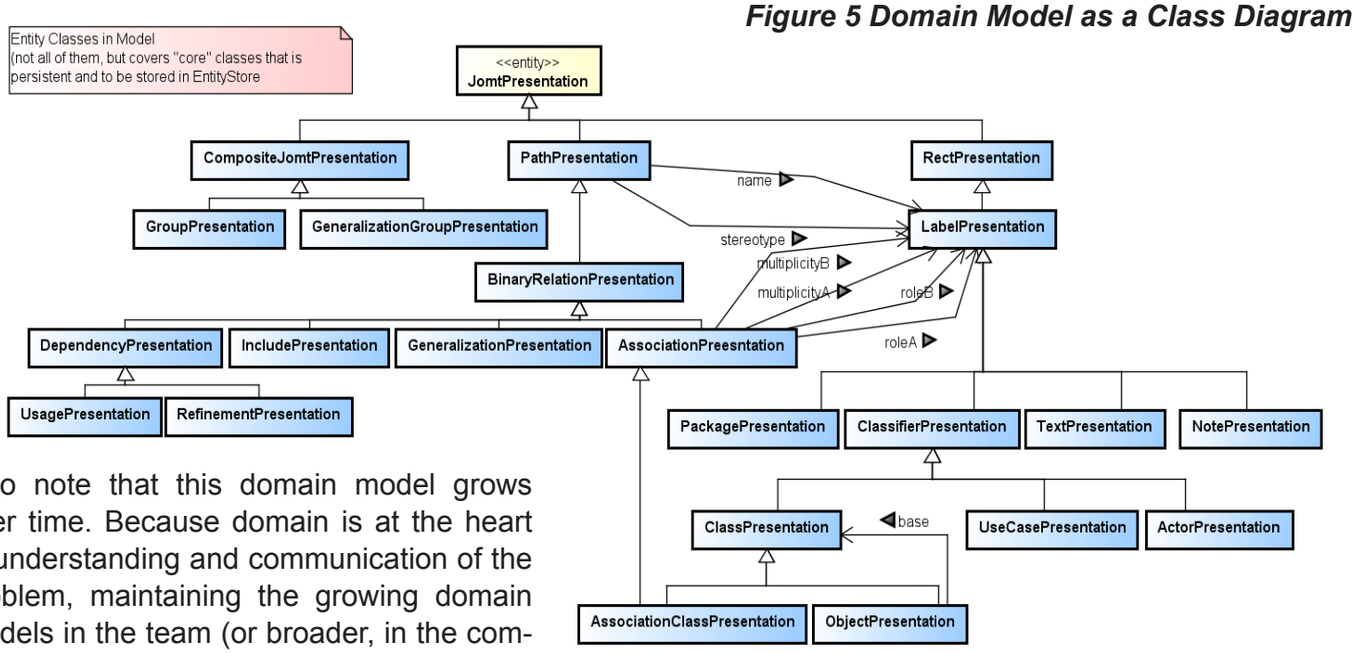
“Dependencies” are often expressed in this diagram between packages to avoid unwanted couplings or circular dependencies. From an architectural view point, inter-package circular dependencies are the evilest problem, and result in harder testing and a longer build time.

2. Domain Models As Class Diagrams or ER/Diagrams

A Domain Model describes the **concept taxonomy of the problem space** in which the application works. In the human communication level, the vocabulary of this domain model is to become the “Ubiquitous Language”[Evans04] used in the whole stakeholder community including the users, domain experts, business analysts, testers and developers.

In the programming level, the Domain Model is also essential for selecting names of programming constructs such as classes, data, methods, and other conventions. A large part of the concept taxonomy (often called “entities”) is mapped into a persistent data structure in the database and often has a longer life than the application itself. Typically, the domain model (or entities) resides in the “M” package in the logical architecture if you choose an “MVC” architecture for your application. In a RubyOnRails type of applications, an ER diagram is more suitable for expressing an domain model because it is tied more directly to relational databases.

Figure 5 Domain Model as a Class Diagram



Also note that this domain model grows over time. Because domain is at the heart of understanding and communication of the problem, maintaining the growing domain models in the team (or broader, in the community) is one big topic that is fully discussed in Eric Evans's DDD (Domain-Driven Development) [Evans04].

Figure 5 is an example of a Domain Model expressed as a class diagram which presents the domain in one picture.

3. Key Use Cases as Use Case Diagrams and Sequence/Communication Diagrams

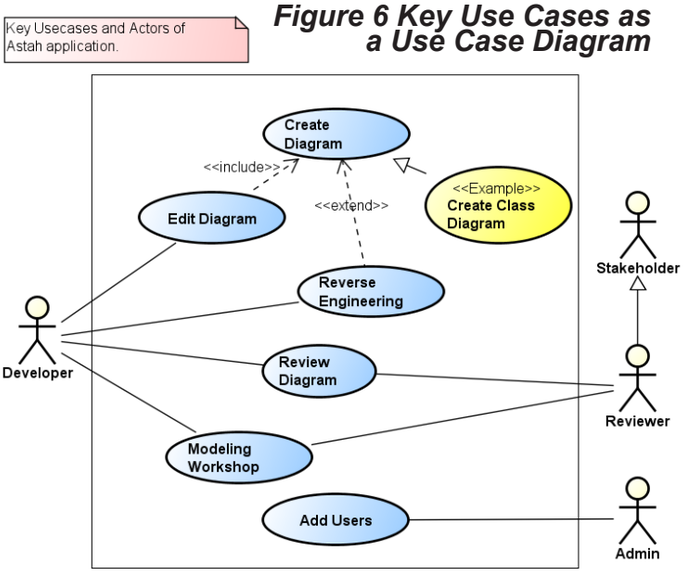
Key Use Cases are **typical usages of the system**, from the user's viewpoint. There are two reasons why we include them in the "Keeps". The first is that developers often go into the solution and forget who the users of the system are and what they want to accomplish with the system. Use Cases help them recall the users' viewpoints and are a good way to have conversations with users, as other documents are usually difficult for users to understand.

The other reason is that Key Use Cases, and their mechanics described as sequence or communication diagrams, work as educational examples for developers. They describe how several objects in the system in different tiers in the architecture work together to accomplish the user goal. Draw a concrete example of a vertical slice from UI to database and illustrate how you implement the Use Case in the architecture.

The Key Use Cases don't have to be complete or cover all situations. Just pick the typical ones and keep them simple.

Figure 6 is an example Use Case diagram which shows the typical users and usage

Figure 6 Key Use Cases as a Use Case Diagram



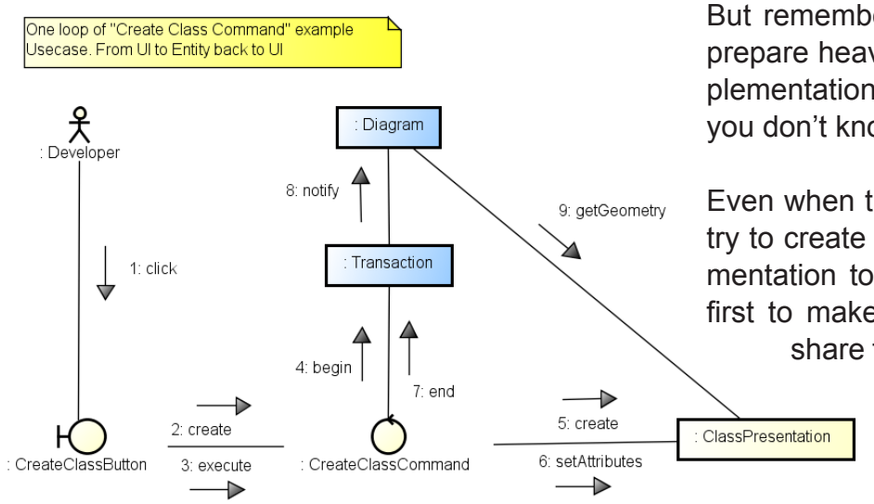


Figure 7 Key Use Case Mechanics as Communication Diagram

of the system. It doesn't need to be comprehensive, but should capture the context of the system. The yellow Use Case ("Create Class Diagram") is chosen as an example Use Case, and the design breakdown is diagrammed as a communication diagram in Figure 7 . With this example, the team can share an understanding of how the Architecture and Domain Model (diagrammed in Figure 4 and Figure 5) actually work to accomplish the features described as Key Use Cases. See Figure 3 for the relationship.

You can use tools to draw these diagrams to make maintenance easier, and to print them on a large piece of paper to have posted on the wall. The wall will then become a discussion place of modeling workshops (as I will soon discuss in a later section).

Scaling

"Rather than divide and conquer, an XP team conquers and divides."

– Kent Beck

With a small team of less than 10 developers, you may not have to maintain any models next to the codebase. As the development scales to multiple teams, you will need to get more benefits from modeling.

But remember, don't invest too much time to prepare heavy documentation (with ZERO implementation) just to hand it over to someone you don't know.

Even when the team gets bigger, you need to try to create a thin vertical slice of your implementation to accomplish the Key Use Cases first to make an architecture seed, and then share the knowledge with sub teams using the working code and "Keeps" models. In other words, don't try to "divide and conquer" by dividing the problem on the desk and throwing the specification over the wall to make sub-teams conquer. [Larman10]

Below, I describe how multiple teams communicate the "Big Picture" using the "Keeps." The first "conquer" should be tried by less than 10 people in one team called the "Tiger Team" at one location. After the first conquer, all the "Keeps" described above can be used as good documentation to communicate the understanding of the system. In Sprint 1, the Tiger Team conquers the Key Use Cases first to establish the first "seed" architecture and makes version 1.0 of the "Keeps" as SAD (Software Architecture Document).

Think of these models not as a specification, but as common ground to create understanding. And again, do not just hand the document over to the sub teams.

The best way to communicate design intention and create shared understanding is to **facilitate a Modeling Workshop** with the sub teams as displayed in Figure 8, on the next page.

In the modeling Workshops, one member of the Tiger Team (Ken, in Figure 8) first explains the SAD and walks through the models. With casual Q & A's, he communicates the core ideas and the structure of the system. He can use the Key Use Cases to explain how the system

components works together to accomplish a user goal. And he jointly designs one or two of their Use Cases with them, using “Temps” models and maybe pair-programming.

Don’t try to make the SAD complete. Use workshops as a way to establish a common understanding by having a rich conversation and without handovers.

An important part of the sub-team workshop is feedback. In Figure 8, Ken in Sub Team 1 and Tom in Sub Team 2 bring the feedback to the Tiger Team and discuss with the other members how to make the “Keeps” models better. The image below is a printed diagram with notes during the workshop. It includes notes for understanding and also notes for feedback.

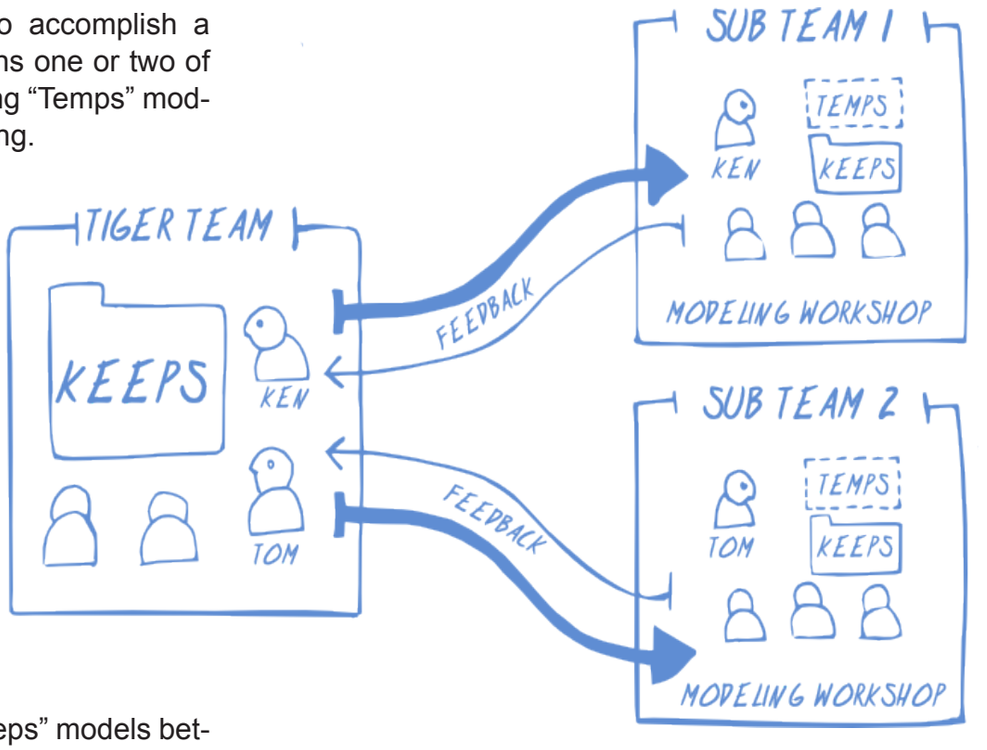
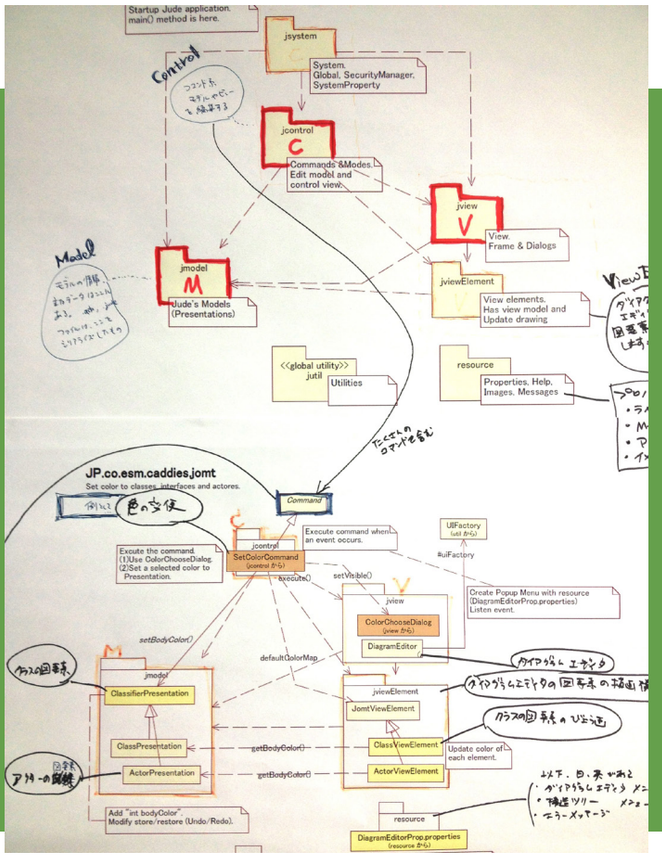


Figure 8 Tiger team and Sub teams



Hold workshops repeatedly. Grow understanding by “modeling,” not by the models. Remember, use “model” as a verb and “Model to have a conversation” [Larman10].

Modeling Tips

With the ideas and experience described earlier, here are some final tips for you to use in daily modeling sessions or workshops.

- **“Reverse and Model”**

A lot of UML tools support “reverse engineering” features that visualize the codebase in a just-in-time manner. Some of them have a nice “Drag&Drop” from source code and even from Github repository URLs. With the packages and classes which you reverse-engineered from the codebase as a foundation, you can start casual modeling not only with the “Keeps”

Acknowledgement

I'd like to thank Hiroki Kondo and Alex Papadimoulis for comments and Ben Linders and Scott Reece for reviewing and editing my article. Special thanks to Craig Larman who first described the importance of modeling workshops (or "model" as a verb) and spent his time on an airplane to give me fundamental suggestions on this article

models, but also with models made directly from the codebase.

- **"Print and Draw"**

As described earlier, good interactive modeling workshops are facilitated with big paper posters on the wall (or on the table) and by having the conversation with them. Draw notes and comments directly on the printed diagram

- **"Projector and Whiteboard overlay"**

Another way of sharing models in workshops is using an overhead projector and a whiteboard together to simulate "Print and Draw". Use a overhead projector to project the "Keeps" on a whiteboard and draw comments or put sticky notes on it.

Conclusions

In this article, I explained how modeling fits into an Agile development framework like Scrum, and proposed the models you could keep throughout the lifecycle of the product. And I recommended facilitating a modeling workshop to communicate design intentions and to establish a shared understanding of the system. These practices become more important when the team scales into many sub teams.

Further Readings

The discussion of design in the Agile context is old. See Martin Fowler's and Jack Reeves's classics.

- Martin Fowler, 2004 , "Is Design Dead?" http://martinfowler.com/articles/designDead.html
- Martin Fowler, 1997 , "The Almighty Thud" http://martinfowler.com/distributedComputing/thud.html
- Jack Reeves, 1992 , "What is Software Design?" http://www.developerdotstar.com/mag/articles/reeves_design.html

The concept of Agile Modeling is first discussed in the book "Agile Modeling". And 3rd edition of "Applying UML and Patterns" covers the topics again.

- Scott Ambler, 2002, "Agile Modeling", John Wiley & Sons Ltd. http://www.amazon.com/Agile-Modeling-Effective-Practices-Programming/dp/0471202827/
- Craig Larman, 2007,"Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development" http://www.amazon.com/Applying-UML-Patterns-Introduction-Object-Oriented/dp/0131489062/

Another InfoQ article that addresses the same problem and context. Still waiting for the part two.

- Lee Ackerman, 2011, "Agile Modeling: Enhancing Communication and Understanding, http://www.infoq.com/articles/agile-modeling-part-one

Wider topics on Agile and Architecture.

- Bill Opdyke, Dennis Mancl, Steve Fraser, "Architecture in an Agile world, 2010", SPLASH workshop http://mysite.verizon.net/dennis.mancl/splash10/

Thanks for reading, learn more about Agile modeling and Astah on our site.

http://astah.net