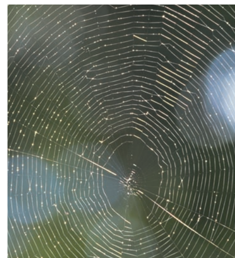
Avoiding the Software Model-Clash Spiderweb

Barry Boehm, Dan Port, and Mohammed Al-Said
University of Southern California

The authors offer two techniques for avoiding the conflicting assumptions that often snare software projects in a costly and time-consuming spiderweb.



Analysts frequently describe troubled projects with the tar pit metaphor used so effectively in Fred Brooks's *The Mythical Man-Month* (2nd ed., Addison-Wesley, Reading, Mass., 1995). We have found a similarly effective metaphor: Think of a troubled software project as an insect caught in a spiderweb of sticky constraints, trying desperately to break free before the spider arrives to feed.

MODEL CLASHES

Our experience and studies of software-model clashes confirm that every software project is unavoidably confronted by a spiderweb of potential model clashes derived from the success models of the project's key stakeholders. These stakeholders usually include the software system's users, acquirers, developers, and maintainers. Additional key stakeholders can include venture capitalists, marketers, proprietors of mutually developed interoperating systems, and the general public when issues such as safety and privacy arise.

A model clash reflects an inconsistency among the assumptions of the various process, product, property, and success models your project uses to guide its progress. Developers use a myriad of process models to create software. Among the most popular, the waterfall model demands the sequential determination of the system's requirements, design, and code, while the evolutionary development model defers the full definition of future increments in favor of developing an initial core capability. Other models include incremental development, spiral development, rapid-application development, adaptive development, and many others.

Product models include various ways of specifying operational concepts, requirements, architectures, designs, and code, along with their interrelationships. Property models define the desired or acceptable level and permissible trade-offs for project factors such as cost, schedule, performance, reliability, security, portability, evolvability, and reusability. Success models include correctness, stakeholder win-win, business-case, or other approaches such as IKIWISI (I'll know it when I see it). Users frequently choose this last model when developers ask them to specify their user-interface requirements.

The underlying assumptions of these various models often conflict. For example, let's look at the key assumptions underlying the sequential requirements-first waterfall model:

1. Participants can determine all requirements in advance of implementation.
2. Requirements have no unresolved, high-risk implications—such as risks associated with commercial-off-the-shelf software choices or the effects of cost, schedule, performance, security, user interface, and organizational issues.
3. Participants well understand the right architecture for implementing the requirements.
4. Requirements match the expectations of all the system's key stakeholders.
5. The requirements' nature will change little during development and evolution.
6. Deadlines allow enough calendar time to proceed sequentially.

When one of these assumptions proves false or contradicts another assumption, a project using the waterfall model will likely run into serious trouble. For example, assumption 1 stipulates that participants can determine requirements in advance of implementation. A project staff that believes this—or that collects progress payments by delivering a complete requirements specification under a waterfall-model contract—will formalize user-interface formats and behaviors as ironclad requirements specifications, then build the system to those specifications. This approach frequently results in a delivered system with a user interface that fails the user's IKIWISI success-model test. Inevitably, the client either rejects the system or demands a major rework effort that is both expensive and time-consuming.

Assumptions 2, 3, and 4 frequently conflict with property-model assumptions if a software acquisition hastily locks the project into unrealizable property-level contract requirements.

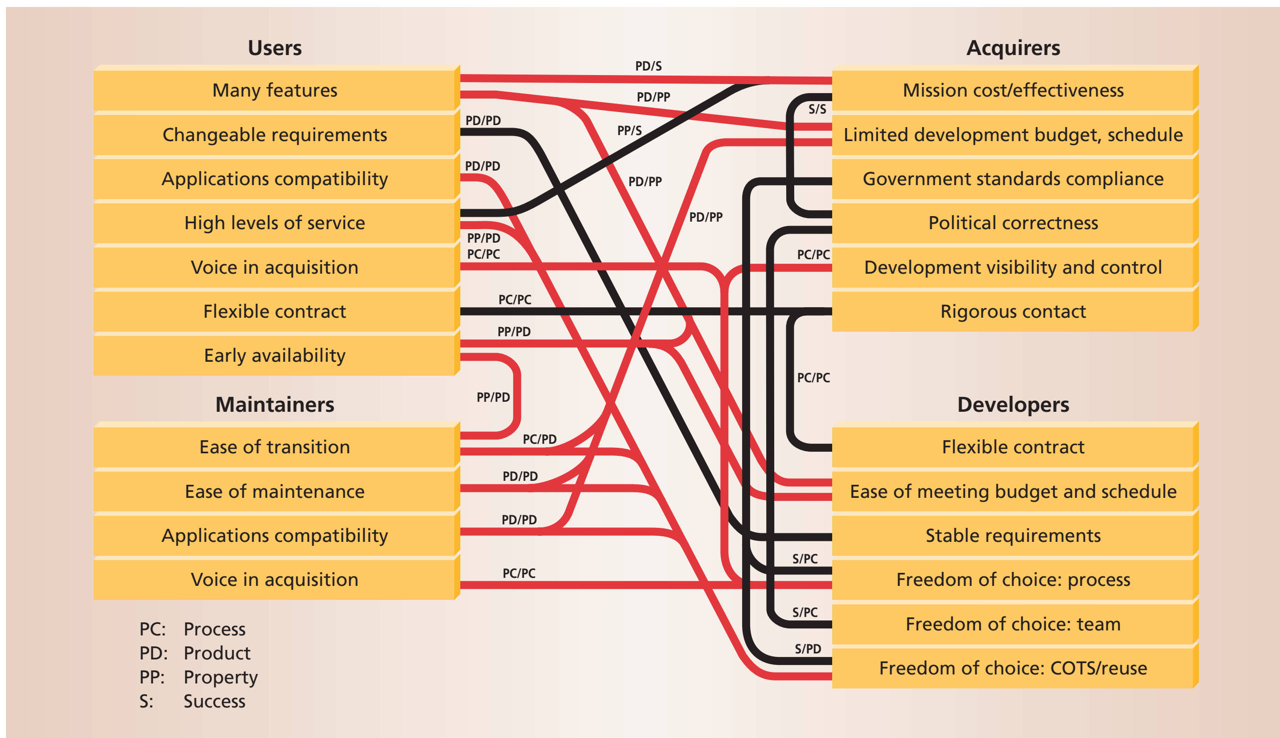


Figure 1. Model-Clash Spiderweb diagram. The red lines show model clashes from the MasterNet system.

MODEL-CLASH SPIDERWEB

Figure 1 shows the Model-Clash Spiderweb, which reveals that potential model clashes occur frequently. Such clashes result from potential conflicts among the most-frequent success models associated with the most-common project stakeholders: users, acquirers, developers, and maintainers. The left- and right-hand sides of Figure 1 show these most-frequent success models. For example, users want many features, such as freedom to redefine the feature set at any time, compatibility between the new system and their existing systems, and so on.

However, the Spiderweb diagram shows that these user success models can clash with other stakeholders' success models. For example, the users' "many features" success/product model clashes with the acquirers' "limited development budget and schedule" success/property model, and with the developer's success/property model, "ease of meeting budget and schedule."

The developer has a success/product model, "freedom of choice: COTS/reuse" that can often resolve budget and schedule problems. But the developer's choice of COTS or reused components may be incompatible with the users' and maintainers' other applications, causing two product/product model clashes. Further, the developer's reused software may not be easy to maintain, causing a property/product model clash with the maintainers.

Clash of models: MasterNet

We have been developing the Model-Clash Spiderweb by analyzing many failed-project examples. The MasterNet system, described in *Software Runaways* (Robert Glass, Prentice Hall, Upper Saddle River, NJ, 1998), provides an excellent case in point.

In the early 1980s, Bank of America chose to develop the MasterNet system, which would update and automate the online generation of monthly statements for the bank's trust accounts. Initially, the project received a $22 million budget and a two-year completion schedule. Project developer Premier Systems tried to forge MasterNet by scaling up an existing small-trust system. Their misguided effort took five years, cost $80 million, and delivered a system that Bank of America rejected.

Worse, the project tarnished BofA's reputation, causing some of its major customers to lose confidence in the bank. The total number of institutional accounts dropped from 800 to 700, while managed assets shrank from $38 billion to $34 billion. The red lines in Figure 1's Spiderweb diagram include these model-clash relationships found in the MasterNet project:

- *Product-Property.* The MasterNet users' product model of many desired features resulted in 3.5 million lines of code, which conflicted with the BofA customers' property model of limited budget and schedule. This conflict led to the large overrun in budget and schedule.
- *Property-Product.* The customer's budget and schedule model assumed stable requirements, but the users' and developer's model of frequent feature changes also contributed to the large overrun.

- *Product-Product.* The developer's model of product needs provided features, such as full customer access, that matched poorly with real-user needs such as accurate and timely reports.
- *Product-Property.* The developer's model of COTS-choice product platform—Prime—offered inadequate performance and reliability to meet users' desired level of service models. Even after several upgrades, the Prime system suffered repeated performance overloads and crashes.
- *Product-Success.* The developer's model of COTS-choice product platform also conflicted with the users' and maintainers' applications-compatibility product model, as BofA had relied exclusively on IBM hardware and software up to that point.
- *Process-Property.* The maintainers' tightly scheduled transition process model conflicted with the testing delays brought on by the customer's limited development budget and schedule model.

By studying these and similar examples, we devised ways to avoid such costly conflicts.

AVOIDING THE MODEL-CLASH SPIDERWEB

Two techniques can help your project avoid a sticky death. One technique diagnoses model clashes in projects already under way. The other technique, when applied to new projects, shows how to avoid such clashes altogether.

Diagnosis

To use the Spiderweb as a proactive review checklist, consider the following large-project review scenario. You arrive at the project location at 8:00 a.m. and receive a huge stack of project plans and specifications for review. You're also given an agenda full of presentations that address various project aspects, followed by a discussion of your review recommendations at 4:00 p.m. What should you do to give the project review the most value?

We've found that the Model-Clash Spiderweb gives you a great opportunity to turn this reactive situation into a proactive one. Request that project management replace its general project presentations with a summary of the project's critical stakeholders and their success conditions. While they are doing this, you can do a quick review of the project's plans and specifications, looking in particular for the project's choices of process, product, property, and success models, and for the potential model clashes among them.

Finding model clashes early usually leads to a renegotiated project definition that avoids later Spiderweb perils.

When project management presents its stakeholders' success models, you can relate them to the Model-Clash Spiderweb and identify potential model clashes, which you can correlate with the potential model clashes you found when reviewing the project's plans and specs. From there, you can devote the rest of the review to a focused exploration of the major potential model clashes and what to do about them.

We have found that this approach produces a more effective project review that usually uncovers at least one and often several serious model clashes. Finding these clashes early usually leads to a renegotiated project definition that avoids later Spiderweb perils, and occasionally it leads to an understanding that everybody would benefit from terminating the project immediately. We have applied this proactive review technique effectively to both small and large projects.

Avoidance

The Mbase—model-based [system] architecture and software engineering—approach can effectively avoid model clashes on a sustained basis throughout a project. Mbase uses a spiral process to explicitly integrate success, process, product, and property models so that project personnel identify and avoid model clashes as a matter of course.

Integration ensures compatibility through the sharing of model information according to the Mbase integration rules. For example, a success model may provide entry and exit criteria for process models. These in turn involve product models as process milestones.

Both product and process models supply evaluation and analysis parameters for property models, such as performance, reliability, cost, and scheduling. Success models also serve as evaluation criteria for product models.

The Mbase guidelines contain criteria for choosing compatible models, techniques for composing models that assure compatibility, and evaluation processes that help detect and avoid model clashes. For more information on Mbase, visit http://sunset.usc.edu/research/MBASE.

Ubiquitous as the Web itself, Model-Clash Spiderwebs present an insidious threat to your management career. Their nature is, however, becoming better understood, and techniques such as Mbase and proactive model-clash reviews show increasing effectiveness in keeping projects out of model-clash trouble. ✷

Barry Boehm *is director of the University of Southern California Center for Software Engineering. Contact him at boehm@sunset.usc.edu.*

Dan Port *is a research assistant professor at the University of Southern California Center for Software Engineering. Contact him at dport@sunset.usc.edu.*

Mohammed Al-Said *is a PhD candidate in the Computer Science Department at the University of Southern California. Contact him at alsaid@usc.edu.*