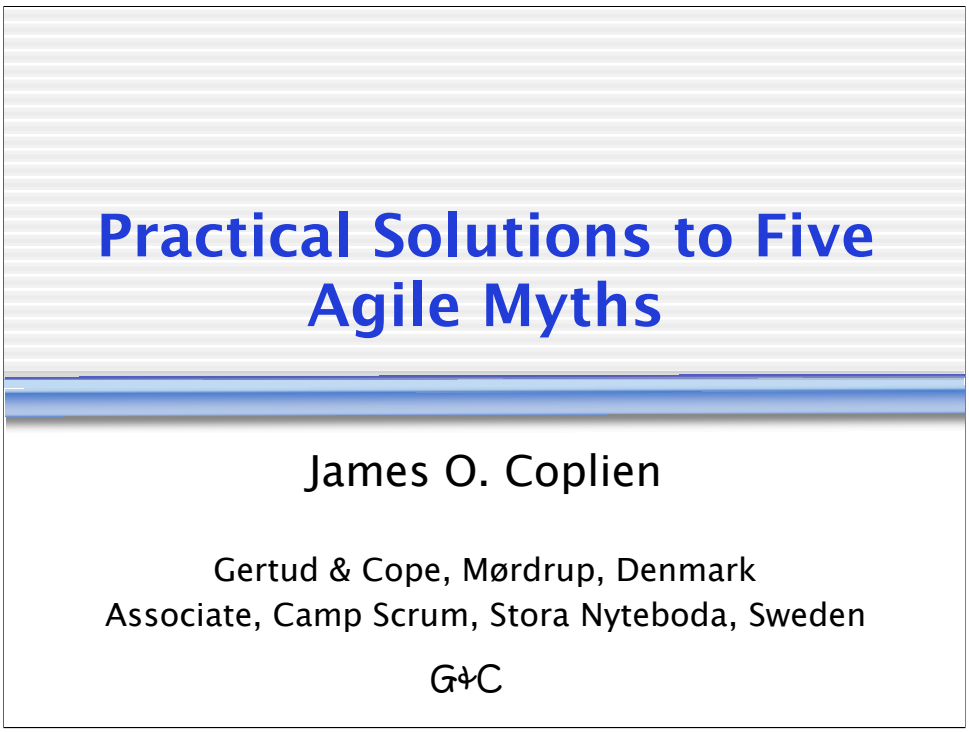
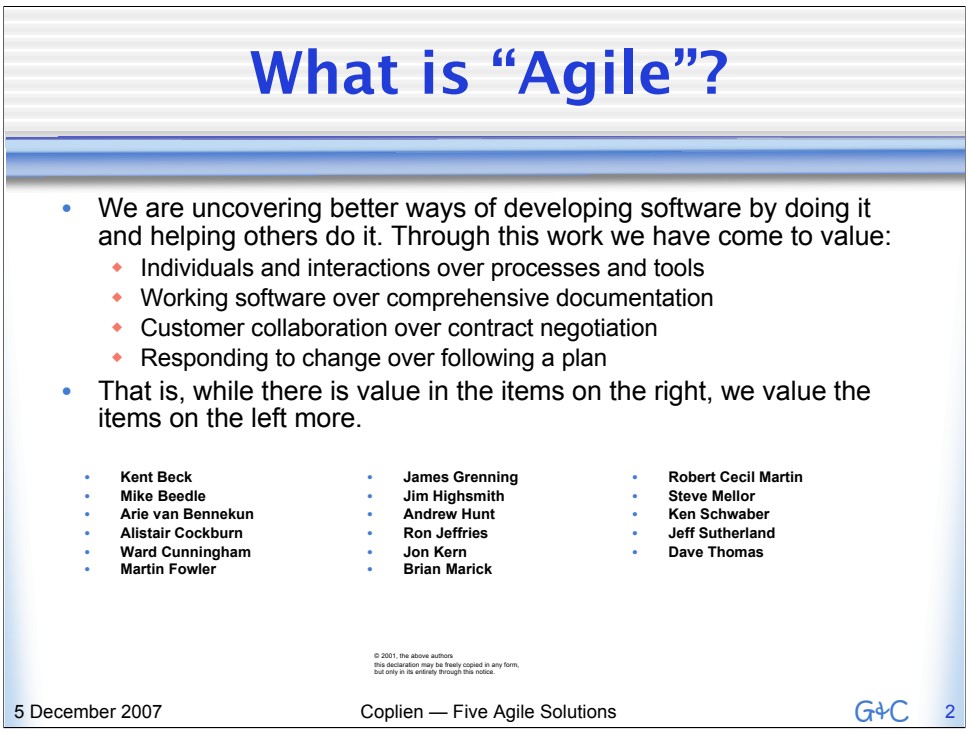


60 minutes. Shoot for 20 slides at 3 minutes each.

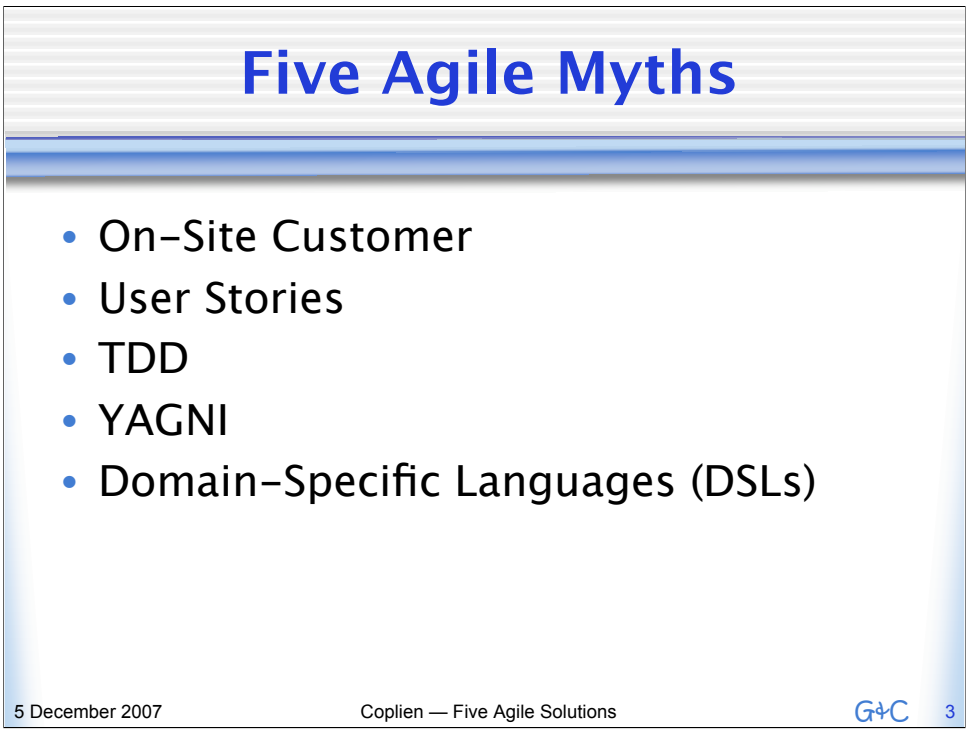
The results are in ... Many ideas in the Agile canon and folklore can actually decrease your velocity or can slowly poison your code. Other contemporary fads have well-known pitfalls that have been forgotten since their last round of popularity. In this talk I look at five of these common practices, why they are harmful, and how to avoid their pitfalls:

- TDD: Use lightweight architecture and an appropriate scope of testing to avoid architecture rot, high test maintenance cost, and usability problems.
- YAGNI: Use Case Slices and lightweight architecture can help avoid being blind-sided
- On-Site Customer: Add a product owner to avoid burning out both the customer and the team.
- User Stories: Instead of deferring detailed scenario development to your development during TDD, use Use Cases to bring the analysis out to the person who matters: the market constituency
- Domain-Specific Languages: Take the value from the analysis and run with it; sometimes, building a domain engineering environment buys you only cost and headaches



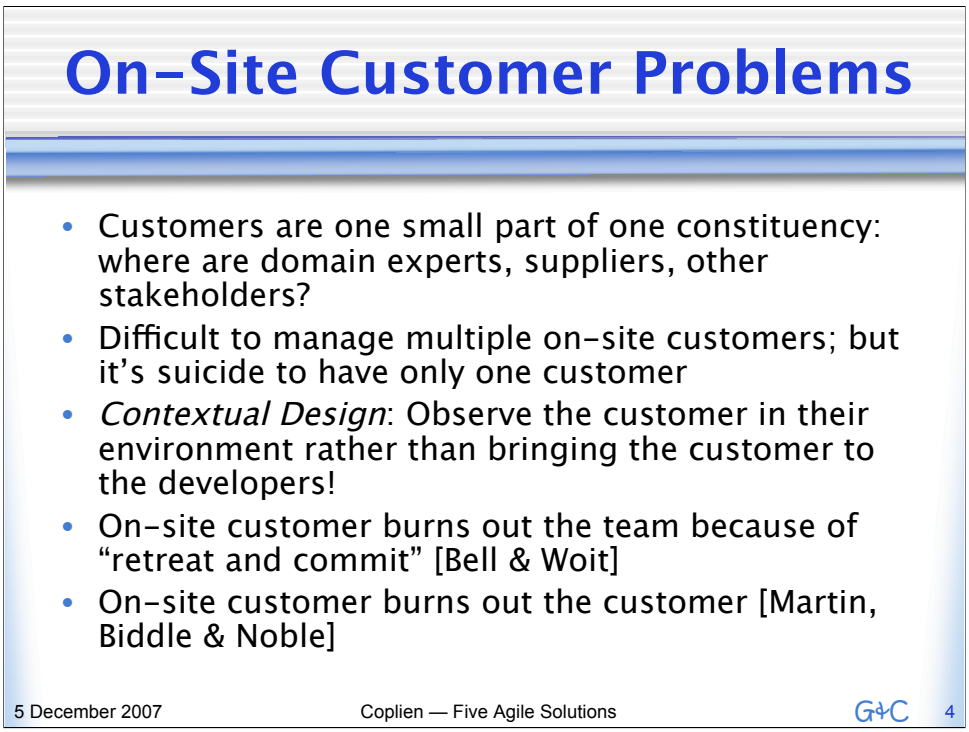
This slide offers the “official” definition of Agile. It is a well-considered and valuable set of considerations with wide public appeal. It is this sense of Agile, rather than any single manifestation of it, that underlies this talk.

I am an advocate of Agile development as defined in this way.



In this talk I'll talk about five myths commonly associated with Agile development. None of these derive directly from the Agile Manifesto; and, in fact, the Manifesto is a brilliant work, and Agile development is in general a great perspective on development. However, some practitioners have found superficial ties from their favorite practices to the Agile values and have pushed them forward as though they were the essence of Agile. In some sense, these practices regrettably have become the core Agile practices. Some of them go against age-old wisdom — and recent experience and some research points to the wisdom in the age-old wisdom.

Here we focus on only five of what is certainly a larger number of myths associated with Agile development: On-Site Customer, the replacement of Use Cases with User Stories; replace of good design methods (and sometimes testing approaches) with TDD; the replacement of good architecture with the YAGNI principle; and the recent resurgence of domain-specific languages. We will explore each of these in turn and, for each, to provide a solution for the misguided pursuit of these practices.



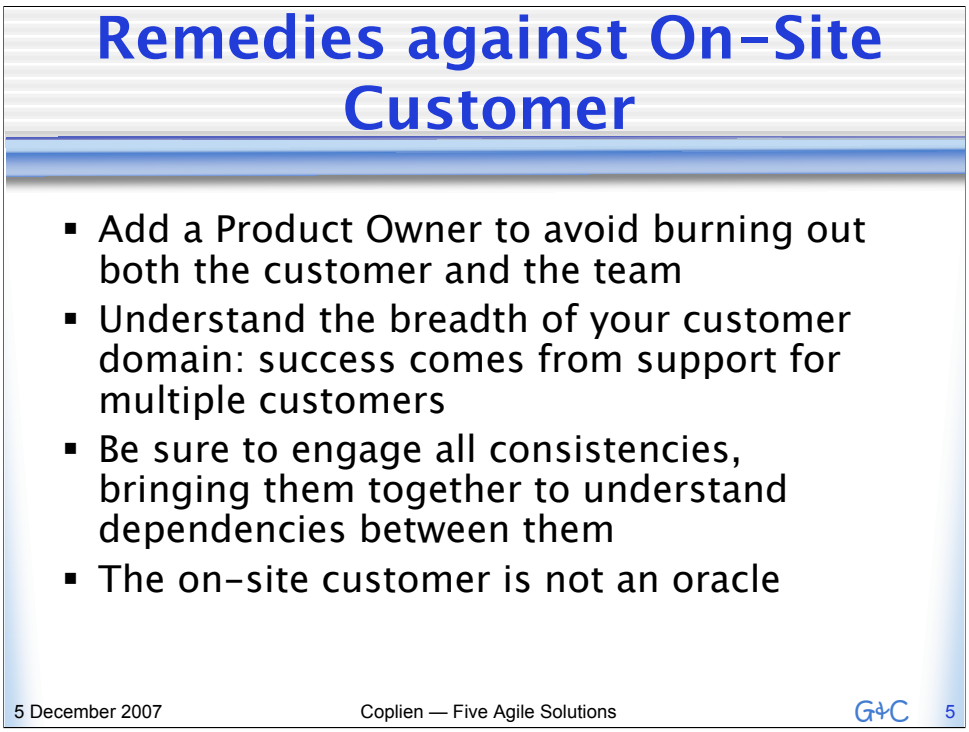
It seems to be an heresy to say not to do customer-centric development. However, having customers talk directly to engineers leads to the retreat-and-commit problem. A customer will determine their minimum agreement criteria and then will ask for more. If the vendor says yes, then the customer is happy. If the vendor says no, then the customer can look good by backing down and asking for less—but still as much as needed. Bell and Woit argue that this problem is particularly difficult when it involves developers who have a personal stake in the accomplishment (their code solves the problem), and they over-commit. That results in missed schedules and burnout. Angela Martin of Wellington notes both that the on-site customer practice is not sustainable,and that it leads to customer burnout. Both of these results are empirical. Detractors say this never happens in an XP environment where trust is complete and where there are no gains, but most projects are not ready for so Utopian a vision. And empirical data contradict that notion.

Contextual Design by Beyer and Holtzblatt advises against bringing the customer to the developer. It's crucial to observe users and customers in their environment. Customers might have an on-site developer, but not vice versa.

Citation: Bell & Woit, "Integration of Social-Psychological Influence Techniques into the XP Negotiation Process", unpublished research. Cited with permission (verbal permission obtained from Kathy Bell on 12 September 2006).

Citation: Martin, Angela, R. Biddle and J. Noble. The XP Customer Role in Practice: Three Case Studies. Proceedings of the Second Agile Development Conference, 2004.

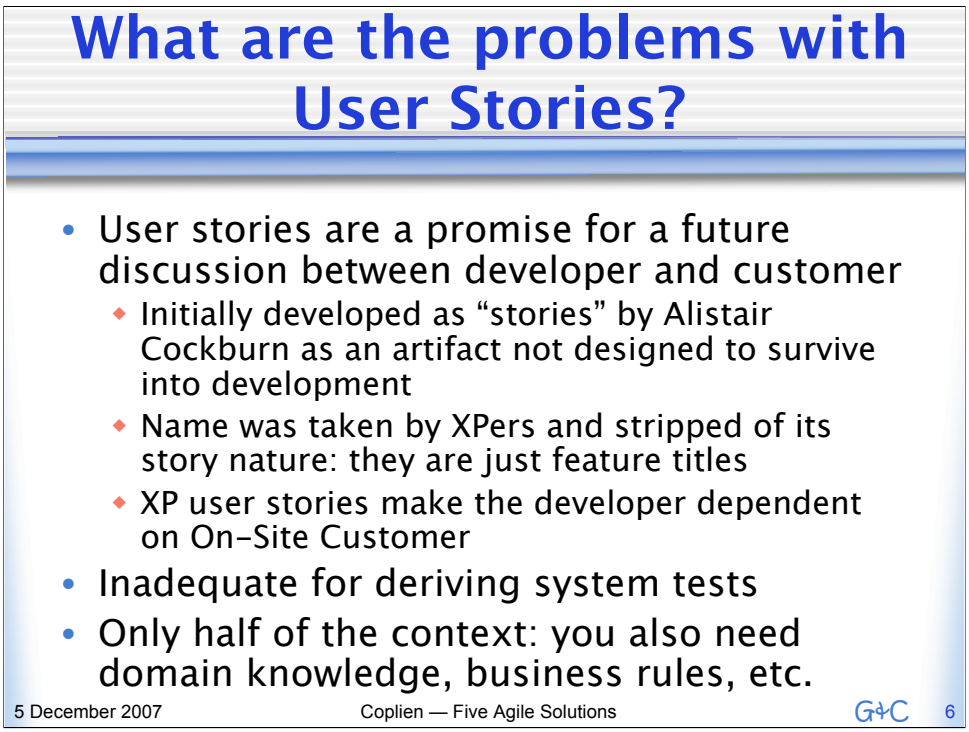
Citation: Martin, Angela. Exploring the XP Customer Role - Part II.



To solve these problems, take a clue from Scrum: Add a Product Owner as a buffer between the customer and the developer. This will temper deliberations and add reason to the feedback loop, instead of driving development with the "desire to please" that developers commit themselves to as part of retreat-and-commit. It will create a communication context that avoids burnout both of the customer and of the developer.

Furthermore, you need to broaden constituency engagement to go beyond a single customer to a single customer. That's how you become profitable in software: to capitalize on your investment many times.

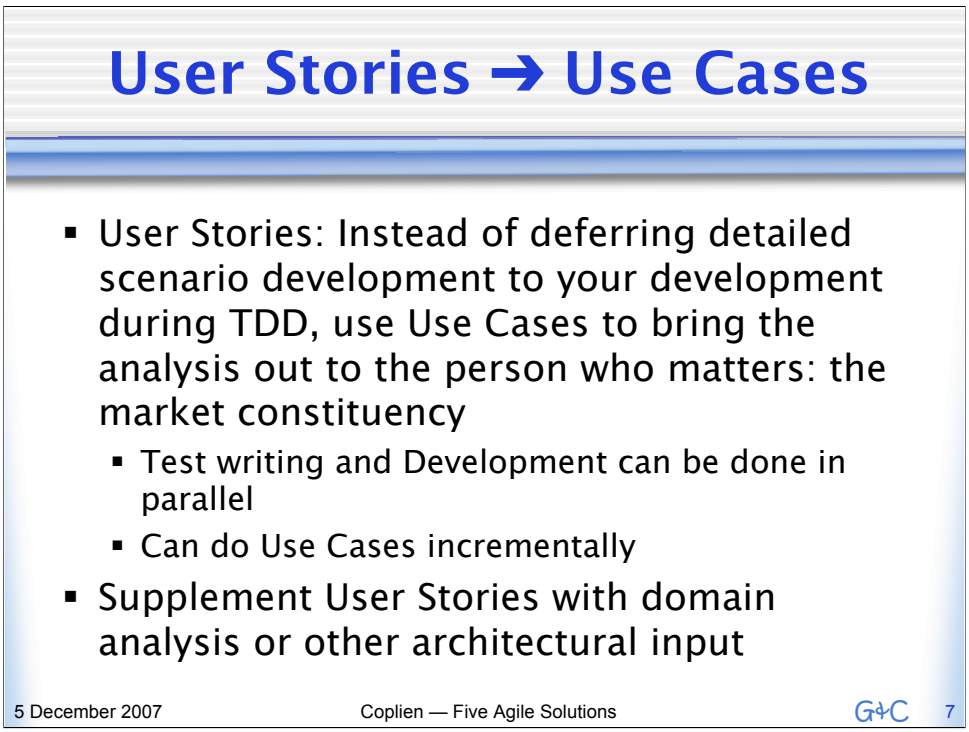
Furthermore, recognize not only that you have stakeholders beyond your customer, but that there are sources of information, help and insight beyond the customer. The on-site customer is not an oracle, but a source to consult.



A User Story is defined as a promise for a future discussion between a developer and a customer. It is not a Use Case: a Use Case is a collection of possible scenarios between a system and its user, together with pre-conditions, post-conditions, triggers, and optionally business rules and other contextual elements that delimit the design. A User Story is just the name of a feature; it conveys little information. A Use Case is powerful enough to drive the authoring of code and tests; a User Story is not.

Historically, the term "User Story" was originally used by Alistair Cockburn to designate a story to get development started. It was a lightweight starting point not designed to survive into development. The term and concept was adapted by XP — which subsequently threw out the "story" part, leaving only a title. Without Use Cases, the developer is left to depend on an at-hand customer (hence its inclusion in XP).

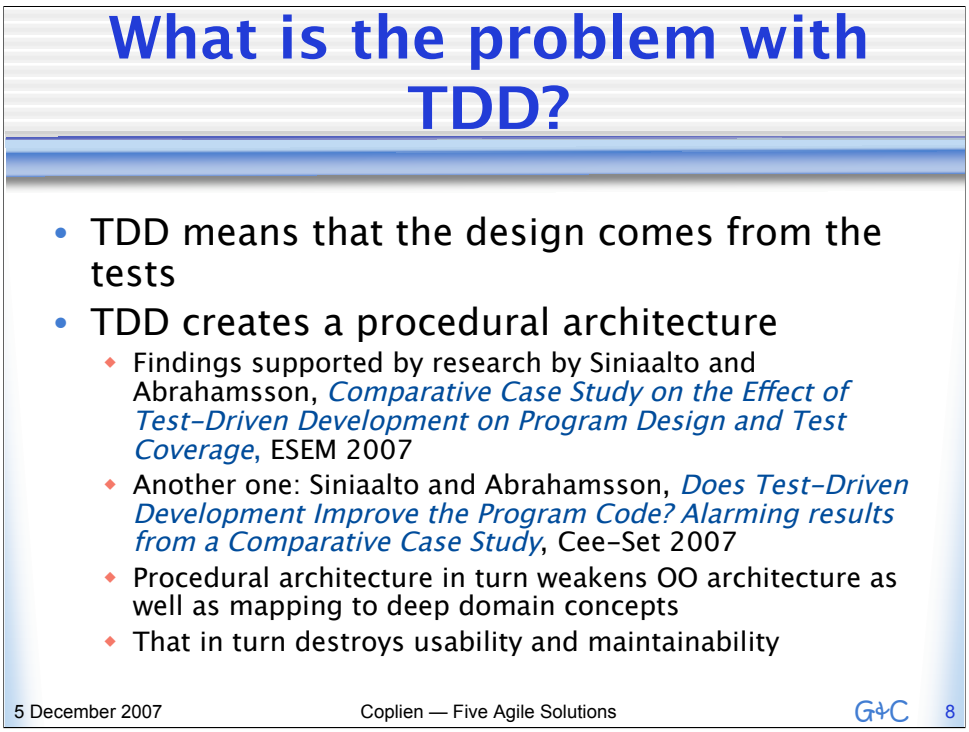
User Stories supposedly drive TDD: the source of design in XP. However, even Use Cases are only half the story: you need deep domain knowledge to derive good classes. More on that later.



Instead of User Stories, use good old fashioned Use Cases. They convey possible scenarios to the extent that they can drive both development and system testing; test development and code development can be done in parallel. Use Use Cases to capture constituency concerns in an evolving record of needs.

You can do Use Cases incrementally: flesh out what you will need for the next release or next few releases. The more Use Cases you can cover, the more feature interactions you will discover, and you will minimize unnecessary rework and refactoring.

Supplement the "what the system does" part of your requirements acquisition with "what the system is" knowledge. We'll talk more about this during our dissection of TDD.



TDD (and closely related TDD) is a design method whereby the tests are written before the code; code is completed to the end of making the tests pass. In TDD, the design comes from the coder's response to the tests. TDD advocates insist that you should put all of your eggs in the test development basket.

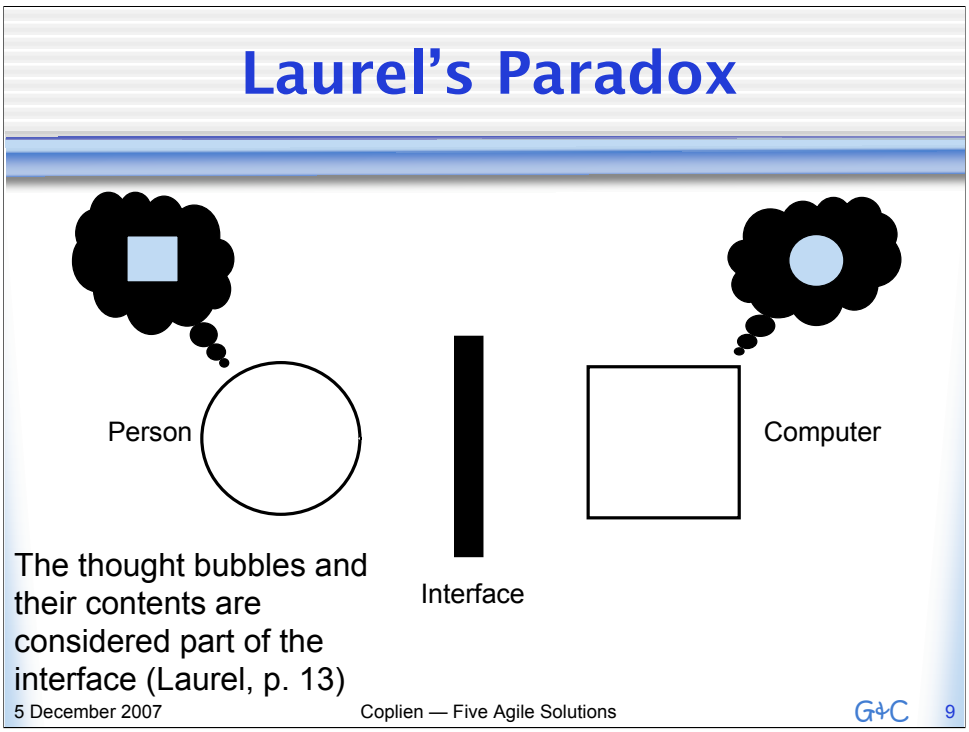
However, most JUnit tests — or any executable tests — focus on "what the system does" as we discussed above regarding User Stories. That sounds like the old days of FORTRAN when procedures were king. That is no longer true in today's interactive systems; to support a good GUI, you need an object model that maps the user's conceptual model of the workspace. TDD empirically weakens your architecture, as found in research by Siniaalto.

Furthermore, this procedural focus destroys maintainability (that is essentially the Siniaalto study) and in fact undermines refactoring.

Even worse, the lack of a good OO architecture seriously undermines system usability, as we will discuss in the upcoming slides.

Cite: Siniaalto and Abrahamsson, *Comparative Case Study on the Effect of Test-Driven Development on Program Design and Test Coverage*, ESEM 2007.

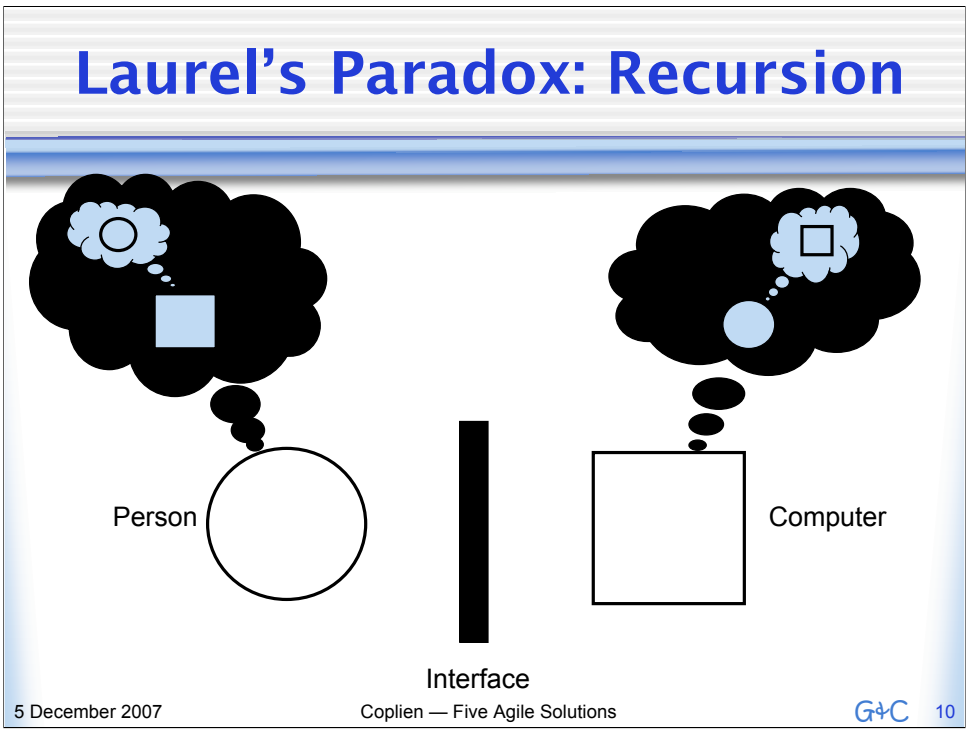
Cite: Siniaalto and Abrahamsson, Does Test-Driven Development Improve the Program Code? Alarming results from a Comparative Case Study. Proceedings of Cee-Set 2007, 10 - 12 October, 2007, Poznan, Poland.



The interface between user and program is a complex thing. It includes, for example, corporate policies (a policy stipulating that all time sheet hours be filled out, whether billable or not, is itself part of the interface). The interface must unify the assumptions of both the user and programmer.

A user always visualizes their business domain in terms of some concepts and objects, and quickly projects that understanding onto the program. In a good program, these two structures are aligned. But they can be aligned only if the programmer has properly captured that domain model. Such a model rarely comes from Use Cases; instead, it comes from domain knowledge.

Citation: Laurel, Brenda. Computers as Theatre. Addison-Wesley: 1993, p. 13.

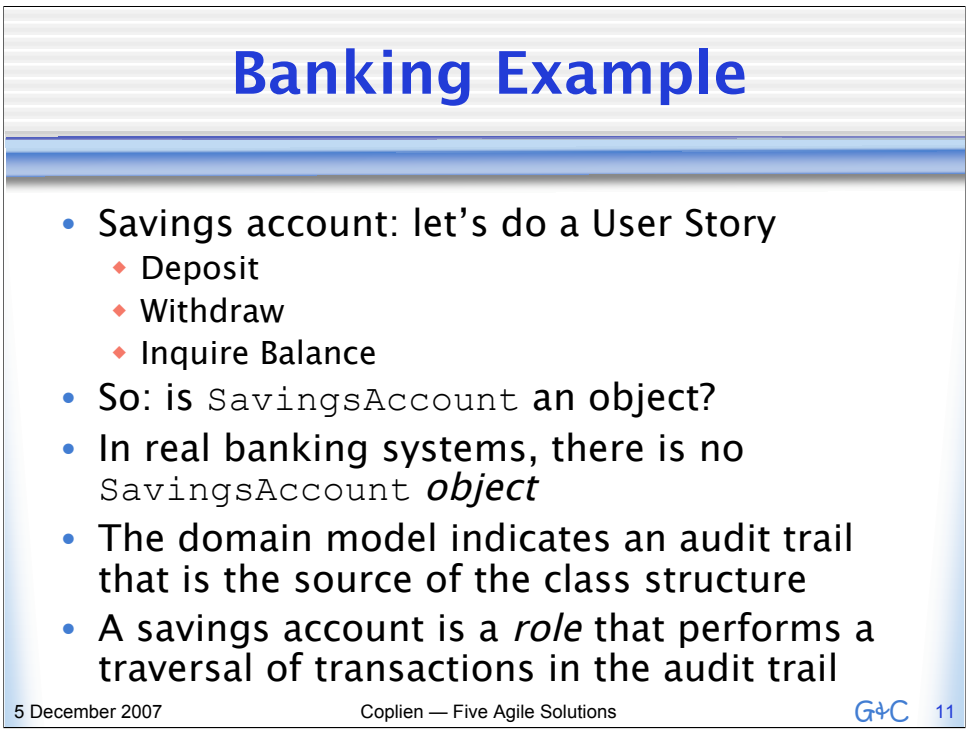


If there is any mismatch between the user expectation of the information model, and what it actually is, you get cognitive dissonance. The user compensates by trying to figure out how the programmer actually modeled the domain in a futile attempt to coax the program into proper behavior. Similarly, the programmer may try to anticipate the user's interaction with his or her creations rather than drawing on the deep domain model, and that complicates the interface by introducing inconsistencies. It is important to have some kind of a "fixed point" to bring these two worlds together.

Ever try to place a picture in the middle of some text using your favorite word processor, and wonder why you couldn't make it go where you wanted it?

If the design is driven by Use Cases, or by tests that reflect these Use Cases, the programmer will create a design — driven by tests — that reflects a structure different than that in the user's conceptual model of the workflow.

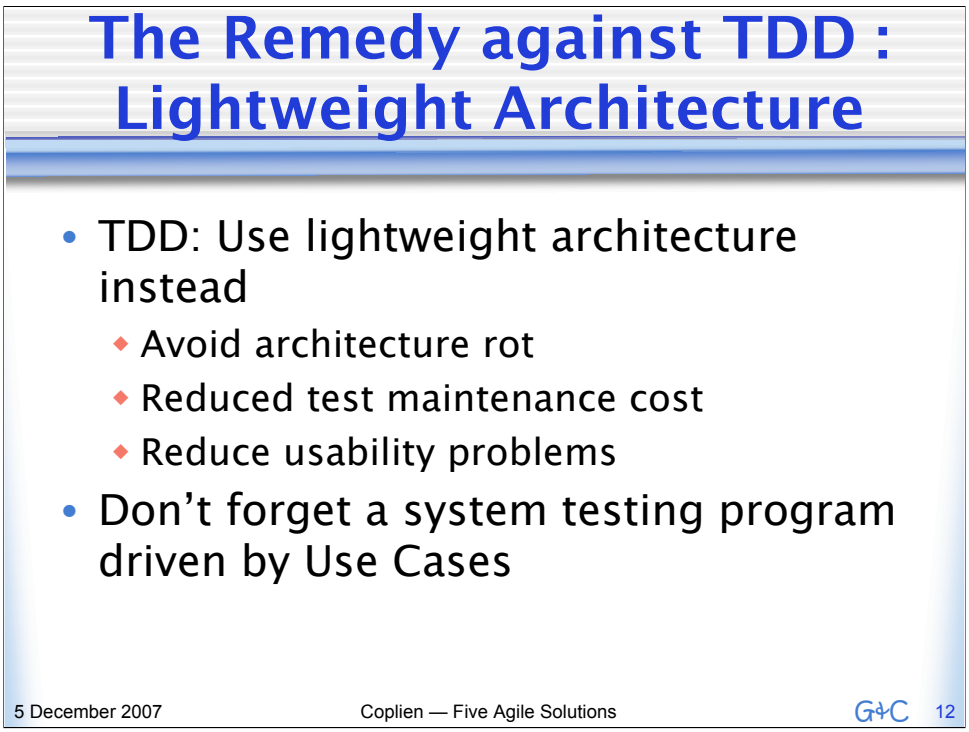
Citation: Laurel, Brenda. Computers as Theatre. Addison-Wesley: 1993, p. 13.



As another example of how tests and user stories fail to drive the programmer to a good design, consider the creation of a simple banking system. From the user perspective, the system is a collection of accounts with simple user operations such as depositing money, withdrawing funds, and inquiring the balance. The question arises: is SavingsAccount an object?

The answer is a a bit complex, but in a good architecture, SavingsAccount would be only a role at the design level and would perhaps be a Java interface in the code. The real object is an audit trail, and the savings account is just a number at any given time that is a sum over some transactions in that audit trail.

In short, TDD can't live up to the needs of design in this example. The problem generalizes, and broad experience suggests that TDD leads to major rework after about the third sprint.



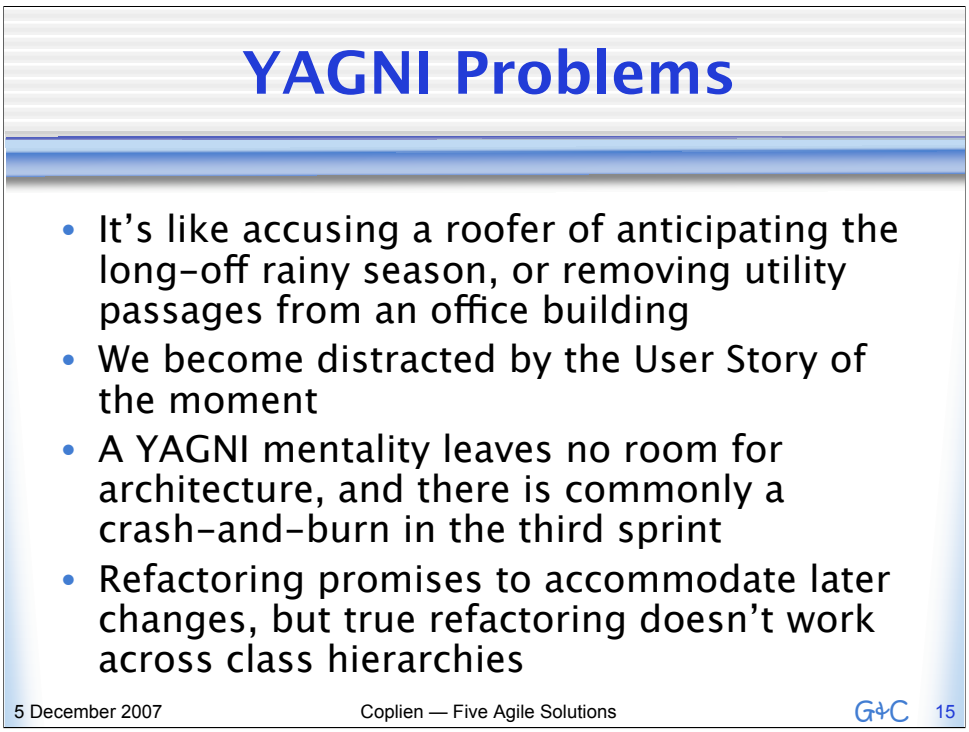
The solution is to use lightweight architecture up front instead of driving the design with TDD. Scope the project well and design domain objects to that business scope. A good domain/architecture basis can avoid many problems of architecture rot, of the test maintenance cost that comes from the high number of tests at the class level that depend on details rather than long-term domain concerns, and of usability. It can help restore the direct manipulation metaphor.

As for testing: don't confuse TDD with a testing program. Put a dutiful testing program in place using good black-box testing techniques.

Cite: Gertrud Bjørnvig, Jim Coplien, and Neil Harrison. The Evil of TDD. Better Software Magazine, November 2007 and December 2007.

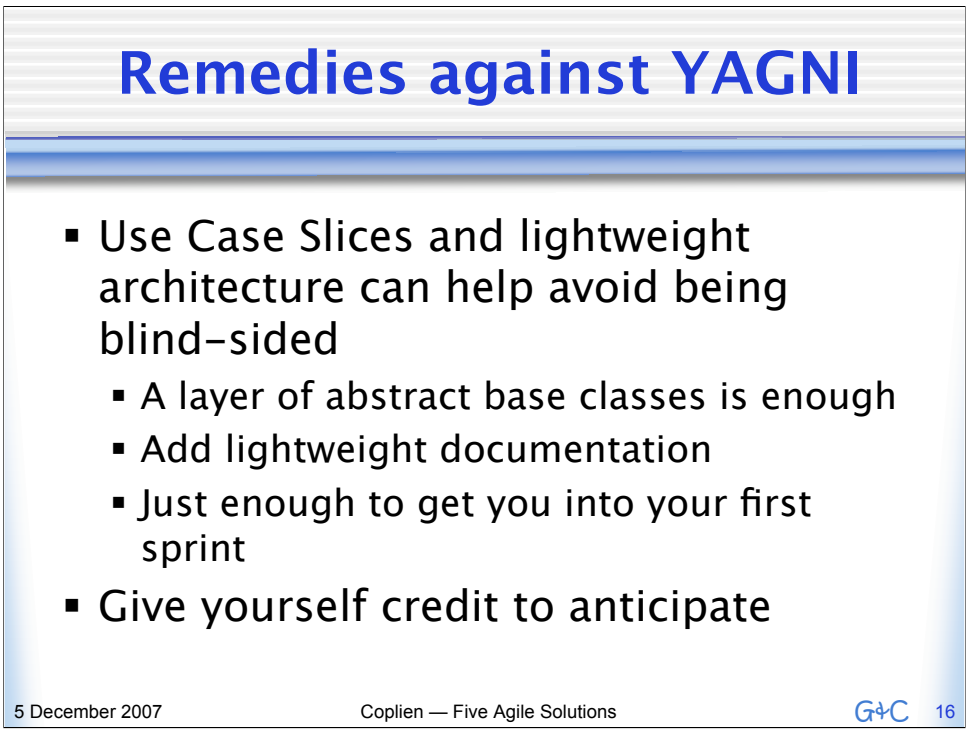
In our idyllic view of the requirements world, we believe we can listen to what the customer says and use that to achieve truth and beauty.

The problem is that the customer doesn't always know what they know -- even when the *do* know it! Beneath the obvious dynamics of Use Cases (we prefer to use Use Cases, Use Scenarios or narratives over XP-style User Stories, which are nothing more than a promise for a future conversation between the customer and developer) we find tacit domain knowledge. Much software structure will follow the structure of this knowledge. And that structure doesn't come from Use Cases.

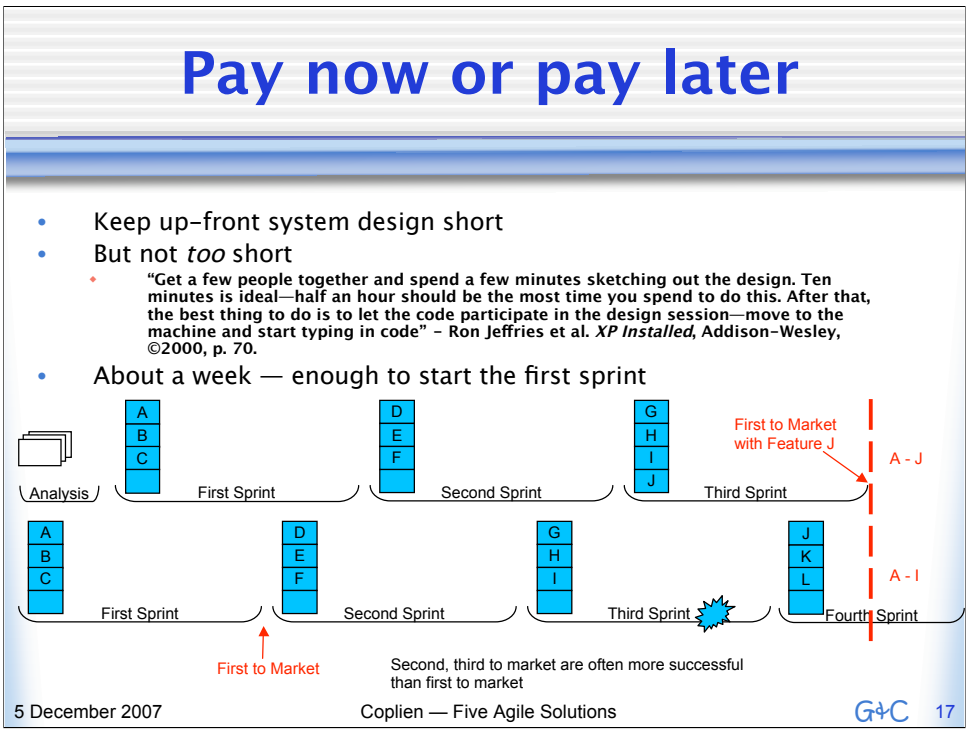


Saying YAGNI is like pleading ignorance. Most YAGNI is an attempt to pretend that we don't know things that we actually know. When we build a house we don't wait until it rains to build a roof, nor wait until there is a new telecom technology to put in utility passages. We think ahead, and we think ahead in terms of the needs of the domain. Good anticipation is the key of good business success. We should capture that anticipation in our architecture.

Of course, if we are distracted by the User Story of the moment, we will build a structure that doesn't evolve well. And refactoring can't save us if the missed concept straddles class categories. We have had extensive experience with this problem in some of our own XP systems in Nordija.



To avoid this problems, give yourself credit to anticipate — if you are steeped in domain expertise and are in contact with domain experts. Capture the domain knowledge in a lightweight architecture consisting of lightweight base classes and some lightweight documentation—all within the Agile values.

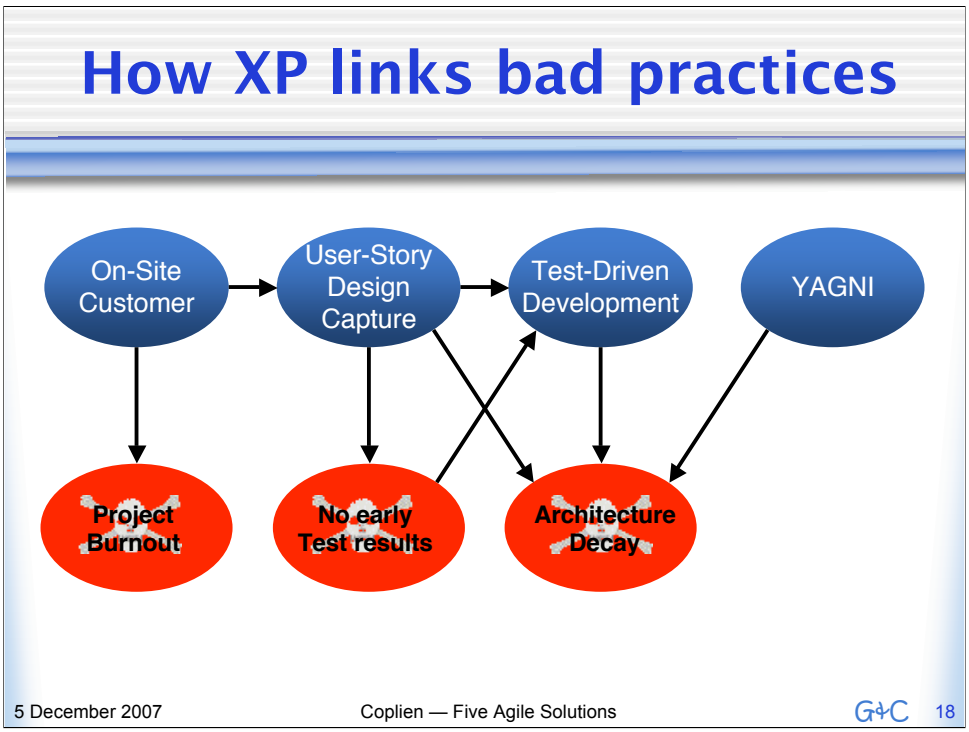


Keep your up-front design short -- about one sprint, if you can afford it. At the end of that sprint, have something that you can evaluate: CRC cards that you can "execute," or a code framework that compiles and that can demonstrate basic sanity. Having that little bit of structure up front will pay off later. The amount of time you spend may depend on your business and domain; in mature domains, or very high-reliability domains such as aerospace, domain knowledge abounds,and takes time to capture. For new accounting techniques related to tax liability for a certain law, there is less prior art to build on.

We have found that many XP projects re-do their architecture within the first three months. That costs time--often, an entire month with no new features, and only with refactoring. That is a month lost to the competition. If every hour you spend doing analysis up front results in more than one hour of savings in refactoring, then the front-end work will eventually be worth it. Popular industry experience suggests that you catch up within three releases (Biggerstaff and Tracz).

Further, industry experience suggests that you can learn much from your competition. Being first to market is hard, and does not correlate strongly to long-term success. It is much better to be second or third to market.

Cite for second to market: "Has Apple Hit the Right Disruptive Notes?" **Strategy and Innovation**, Vol. 3, No. 4, July/August 2005, as Apple did with iTunes, iMovie, and Garage Band.



So far we have looked at four XP practices for which there is a strong litany of trouble. This model shows how these practices build on each other. Early XP underscored these these interdependencies as an argument for the all-or-nothing philosophy of XP.

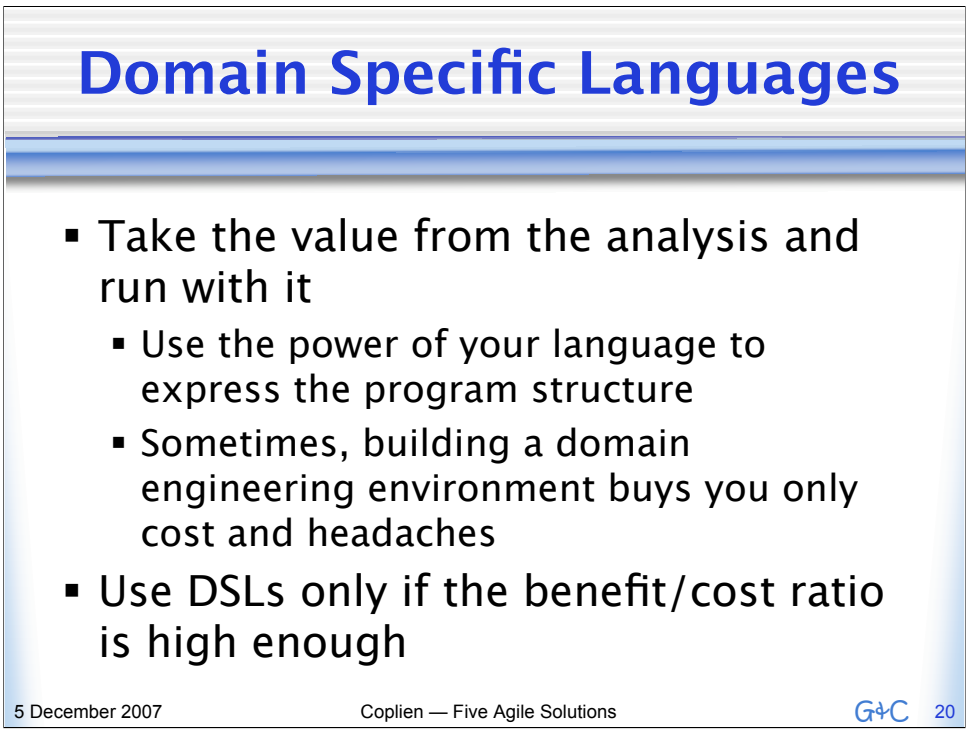
It is more likely that the dependencies arose historically out of a desire to distance XP from the heavyweight practices of Use Case overkill, CASE tools, and architectural formalism. It started with a putting aside of architecture and Use Cases. Use Cases were replaced by stories. Eventually, architecture was replaced by TDD. Other practices such as YAGNI and On-Site Customer would replace the picture later. [Bjørnvig, Coplien, and Harrison]

The argument was that these support each other. In fact these practices propagate fundamental misunderstandings about what works in software development and work together to create burnout, to reduce quality, and to decay the architecture.

Cite: Gertrud Bjørnvig, Jim Coplien, and Neil Harrison. The Evil of TDD. Better Software Magazine, November 2007; Better Software Magazine, December 2007.

Another rapidly rising technique is domain analysis and domain-specific languages as popularized in several recent books. Domain analysis is a great idea and can be the foundation of a good software architecture. However, going all the way to a domain-specific language raises special challenges. It sounds like a good idea, and it's popular because it focuses on the programmer in the same sense that so many Agile practices bring the focus back to the programmer.

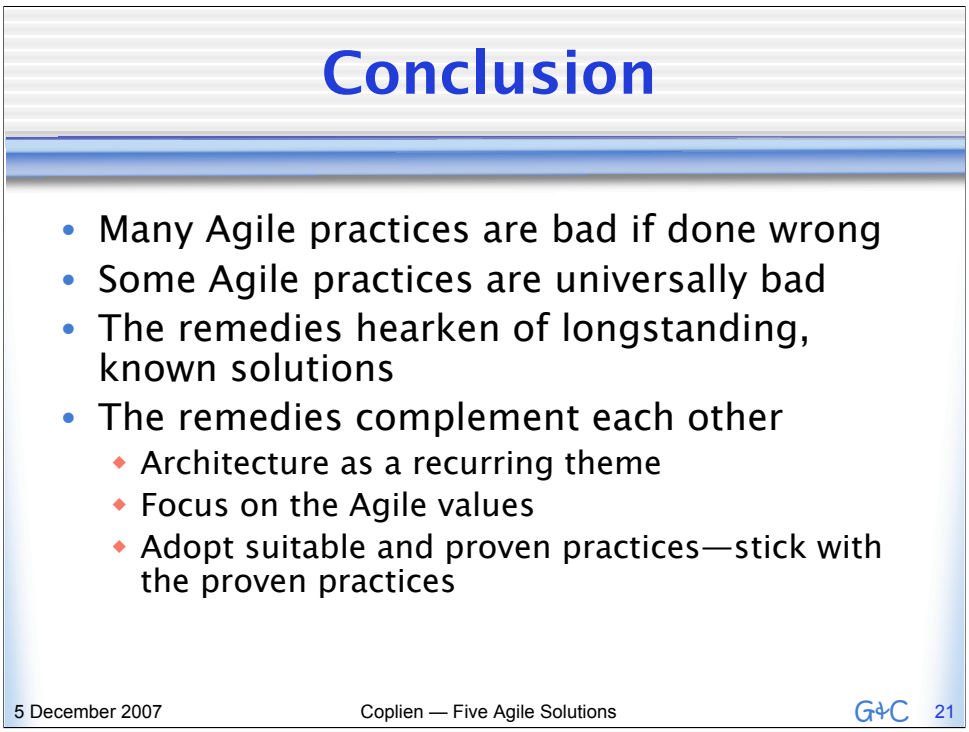
Our experience (at Lucent and Avaya) is that while domain-specific languages sound like a good idea, that they work out only rarely. First, if you build a DSL for each domain in your system, you end up with a tower of Babel. It is difficult for programmers to master one language, let alone several. Together with the understanding comes tool support: while it's fun to build translators, the hard everyday work of building debuggers, configuration management platforms, and other tools is not fun and is often ignored. Last, the world is full of bad application-specific languages— languages that have either been patched badly with the hindsight of practical use, or that have remained unchanged in the light of evolved understanding.



When faced with an opportunity to develop or use DSLs, do a sober cost-benefit analysis. Count the costs of support tools, learning curves, and development; assess the benefits carefully.

Domain Analysis has tremendous value in its own right. Take the value from a good domain analysis and use it to drive your architecture.

Cite: Coplien, Multiparadigm Design for C++. Addison-Wesley: 1998.



Just because something is part of the Agile canon doesn't mean that it's good. There is in fact longstanding experience and theory that describes why many Agile practices don't and can't work. That doesn't mean Agile is bad; it just means that some methodologies have been lured to sexy or simplistic solutions instead of building on good experience, good research, and understanding. It's almost understandable that projects grasp at anything they can. But when these ideas, which in fact can be harmful, reach the level of notoriety as the XP practices have, the widespread damage sets back an entire industry.

Some Agile practices are almost always good—like good communication. Some are good only part of the time—like pair programming. Some, like those focused on here, have very little to substantiate their value. It is O.K. to explore and experiment, but sound development is grounded in proven practices. Many proven practices that pre-date XP and Agile as a buzzword are perfectly consistent with the Agile values, if done in moderation. We have mentioned some of them here: lightweight architecture, Use Cases, and system testing.