Visual Design and Architecture

Presented at 2019 O'Reilly Software Architecture Conference NYC

Ruth Malan
Bredemeyer Consulting

Annotated Presentation Slides

These slides and annotations offer an overview of the why and how of visual design and architecture. The notes are more detailed than a transcript of the talk; they follow the flow created by the slides, but add some more detail here and there, to nuance the discussion further.

Key idea: Sketching and modeling is indeed a way to be agile – when it allows us to learn with the cheapest method that will uncover key issues, alternatives, and give us a better handle on our design approach. Not to do this as Big Modeling Upfront, but to have and apply our modeling skills early, to understand and shape strategic opportunity, and to set design direction for the system. And to return to modeling whenever that is the cheapest way to explore a design idea, just enough, just in time, to get a better sense of design bets worth investing more in, by building the ideas out and deploying at least to some in our user base, to test more robustly.

Outline

Introduction

• Motivation (overcoming the anti-modeling undertow created in wake of waterfall/BDUF);

Part i. Design of what the system *is* (system-in-context design; “requirements”)

• Level setting (what architectural decisions are and entail; why architects should be involved/partner);
• How: illustrated (design scopes and views, by example)

Intermezzo: why visual design in engineering?

Part ii. Design of the system (internals): system structure and dynamics (architecture)

• Finding abstractions: responsibilities; heuristics
• Improving/refining abstractions and designing key mechanisms: structure and behavior modeling
• Exploring interplay of design and physical topology: revisit abstractions in light of physical boundaries and (desired) runtime properties
• Improving the design: views and reviews, perspectives,

Wrap up
This talk is about visual design and architecture. That means that while there is a lot about architecture and software design that visual design doesn't address, we're focusing here on visual design, and exploring where and how it applies to architectural design in an Agile context. While Waterfall development relied heavily on visual design because design was, at least in theory, completed before the code was written, in Agile development, design is recast.

Now, many people are saying that code is design. And this isn't new.
In 1992, Jack Reeves wrote an essay published in the C++ Journal, in which he compellingly made the point that the code is the design that gets built, and, importantly, we design in code. When we're programming, we're designing.

This was at the time that object-oriented design in the form of the Booch method, OMT and OOSE, and then UML, were gaining sway. Jack’s insights stuck a note of resonance, and unbeknownst to Jack Reeves, his essay was taken up on Ward Cunningham’s C2 wiki, and was being discussed with some vigor.

Something else happened, that Jack Reeves didn’t intend, and that was that some took him to be saying “the source code is the design” means ‘don’t do design, just code’.” So in 2005 he followed up with another essay, to rectify this take. And he highlighted his point from the earlier essay, that “In software engineering, we desperately need good design at all levels.” And “Designers should use anything that helps.”
And he returned to his point that our sketches or diagrams and models are not the design the compiler builds—the source code is. Noting further that since we have to do the "real design" in code, "we should not be afraid to code our designs as we derive them."

At the same time, the Agile Manifesto was quickly gaining following and causing a sea-change in how we develop software. The first Principle, "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software," establishes the focus on continuous delivery of working software.
What is meant by software design, generally depends on who we’re talking to.

If we’re talking to product owners and user experience designers, design means design of working software—what the system is as experienced by users of the system. And it encompasses determining the capabilities and properties of the system and how it works from a user point of view, determining the avenues, scope, and particulars of user interaction with the system.

In the context of conversations with developers, design typically refers to design of the code, how the code is structured and how the code works (to deliver the capabilities and properties of the system).

The other design surface, and one that we are increasingly paying more explicit attention to, is operations and design for operability and observability — where we learn about the code (system behavior, under load, environmental stresses including component failure, and other demands) and the business (user behavior, patterns and anomalies, etc.).
In Agile, we're advancing both faces of design as we iterate, deliver working increments of value, get feedback, respond, and adapt. We're not just gaining a clearer understanding of the system; the system is evolving. Users adapt to the system, shifting work to the system or doing new things the system makes possible for them, and new possibilities for the system come into view. And using Agile practices like Test Driven Development and refactoring, we advance the design of the code.

[Aside: We have used "right system, built right" and it usefully points to the two (sur)faces. That said, I want to "loosen" or shade the notion of "right." As we design-evolve the system, we learn what the system is becoming, what users and others want from it; but also, the system changes its contexts, raises new challenges and opportunities as it comes to be used in new ways. And so forth. The point is, it's a responsive process — generally imperfectly responsive. We don't have perfect understanding, the process is strewn with trade-offs (read compromises!) and judgment calls (clouded by complexity and uncertainty, in the fog change churns up), and the system and its contexts evolve, so what "right" is, needs to evolve.

"In his book “The Reflective Practitioner," [Donald Schon] took issue with technical rationality, which in its focus on problem solving overlooked or even ignored the issue of problem setting. He understood design as a reflective conversation with a situation, in which the designer frames and re-frames the problem, the process spirals through stages of appreciation, action, and re-appreciation. The unique and uncertain situation comes to be understood through the attempt to change it, and changed through the attempt to understand it.’ — Jean-Pierre Protzen

That is, to further complexify the design matter, we're not just designing these "(sur)faces" of design, but redesigning, in effect, (at least part of) the containing systems. Suffice it to say, at this point, that the design of the development organization, per Conway's Law, is not independent of the system design; if ignored, the organization design will (regardless of architectural intent and expression) impact the system design. The design of the system impacts the responsibilities and actions and choice sets of the containing socio-technical system. Whether we take this into account or not, we're changing the context, so better to be more intentional about how those impacted want things to be, to better shape the system for fit to purpose(s) and to context. We're designing not just the system, but the system-in-context, taking into account that the system is changing the world for users and others for its various contexts of use and impact (ecological impact, too). It makes new things possible which is why we're building it, but has potential negative impacts too, and we want to make more informed tradeoffs and design choices.

We can consider design at different scopes: the system, the organization that builds it, the socio-technical systems that use it, our business and, if our systems are used by customers, their business. The ecosystem? All impacted by design choices, which means we can hope and try to make better choices, by learning how well we're doing, and making adjustments. More on these considerations later.]
So this "the code is the design" point is even more compelling in the light of Agile, than it was when Jack Reeves first made it. "The cake is a lie" meme is well played, when documentation is "obsolete the moment it's written."

Returning to the Agile Manifesto for a moment, the penultimate Principle assures us that "The best architectures, requirements, and designs emerge from self-organizing teams."
But, as Erik Dietrich points out, good design doesn't 'emerge' like a ray of sunshine breaking through the clouds.

"It comes coughing, sputtering, screaming and grunting from the mud, like a drowning man being pulled from quicksand, and the effort of dragging it laboriously out leaves you exhausted."

—Erik Dietrich
Still, as Tim Ottinger points out, when we say "the code is the design," we mean we've done the intentional and painstaking work that design takes. Ideally, on both design (sur)faces—we observe, learn, improve and adapt both the software (as experienced by users) and the code (as experienced by developers).

[Pressure to evolve and adapt the system doesn't just come from building out the system incrementally. The context evolves. The landscape changes, but the system also changes its context. As users and containing systems adapt to the new or extended capabilities of the system, new expectations and demands arise. New demands surface new forces and challenges, and new solutions are sought. With new solutions or capabilities, the system is put to new uses. These cycles of co-learning (symmathesy) and co-adaptation (even co-optation), mean that we need to be moving the design along both fronts—software and code. Not just adding to the code, but treating the code design as a design surface where we also need to observe, learn, and improve the design. When we don't feed design learning back into the code design too, tech debt mounts. When entropy sets in, we've lost agility. See also Lehman's Laws: 1. Continuing Change, and David Woods "Law of Stretched Systems."]
"Code is design" is compelling, for it places emphasis on code as a design medium that turns directly into something we can put in the hands of users, to learn, probe for effectiveness, and improve or adapt. So the point here is to appreciate Agile, and where we are, but also to allow ourselves to reflect. And, indeed, the Agile Manifesto begins "We are uncovering better ways of developing software by doing it and helping others do it," inviting this reflection and disciplined openness to improvement.

Which brings us to the framing, orienting question for this presentation: if all we have is code (including tests), and from the perspective of design, what are we missing? What would help us design better, and/or better express and communicate our design, or facets thereof? Given what we're trying to learn, at this moment we're at, what is the quickest, cheapest, most effective way to explore, surface and address design challenges we face? It might be code, but what else? When and why?

As we do design, can we do more to help ourselves and our teams with the cognitive challenge of designing complex systems, or help us collaborate better on the design, bringing our different perspectives, experience sets, and ideas to the design effort?

We found that modeling too close to the code, duplicates the reasoning we can do in code, and we don't want to repeat that learning. We're looking only for a difference that will make substantive, consequential difference. Being fleet and responsive means avoiding gratuitous, low impact work. We want to only consider visual design to do what we don't do in code better, or anyway.

The code is necessary and sufficient design expression to be built (by the compiler). And any other (design) documentation is neither necessary nor sufficient to be built. Still, the compiler isn't the only player. We might ask if code, alone, is sufficient for humans to design and evolve complex systems. And if not, what helps, and what helps enough to be worth it?

Clearly, given how much undocumented or poorly documented (perhaps because what documentation there was, is now obsolete) code there is out there, delivering value, we have an abundant (by code mass) existence proof of something; but can we do better?

"Programs must be written for people to read, and only incidentally for machines to execute." — Harold Abelson and Gerald Jay Sussman with Julie Sussman. (Structure and Interpretation of Computer Programs)
“instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do” — Donald Knuth

We can and do deliver value doing design in code, and expressing that design only in code (including tests, hopefully). Of course, when legacy systems become entangled and ensnared by coupling and poor design, agility peters, despite, or because of, the malleability of code. Returning us to the question: can we do better?

**What is Design?**

What are we talking about, when we talk about design?

“The engineer, and more generally the designer, is concerned with how things ought to be - how they ought to be in order to attain goals, and to function.”

— Herbert Simon

Better at what? When we talk about design, what are we talking about? In *The Sciences of the Artificial,* Herbert Simon ("ground zero in design discourse" — Jabe Bloom) notes "Everyone designs who devises courses of action aimed at changing existing situations into preferred ones." This characterization is profound, for all its straightforward simplicity. He also said design is concerned with "how things ought to be—how they ought to be in order to attain goals, and to function."
And architecture? Architecture is design. Not all design, but importantly, architecture is design. In particular, "architecture represents the significant design decisions that shape a system" (Grady Booch).

So really we’re talking about decisions, and in a letter to Amazon shareholders, Jeff Bezos made this important distinction between irreversible and reversible decisions, emphasizing that consequential irreversible decisions need to be made with great deliberation and consultation,
for they are like one way doors, and once you pass through, you can't get back.

But most decisions are changeable or reversible, and you don't have to live with the consequences of making the wrong decision that long.
Returning to Grady Booch's characterization: "architecture represents the significant design decisions" where "significant is measured by cost of change."

• Significant is measured by *cost of change*
  • Decisions to reduce cost of change (make reversible)
  • Decisions that have high cost of change (irreversible)

These architectural decisions then, may be significant because they make changes easier (within some range of change). Or they make changes reversible, easier to back out of, at least if the need to do so is discovered soon. Or they may be consequential and irreversible and we need to pay close attention to them.
As Eoin Woods notes, they are the decisions that, if we get them wrong, cause the project to fail.

If a decision impacts identity, mission or defining value, and the very viability of the business, it is strategic. If it would cost substantial resources and time, defer value or require major organizational change to revert and rework, it is strategic—it impacts viability. Likewise, if a decision meaningfully reduces the cost of making changes, enabling our business to be more fleet, adapting and extending its services or product set as opportunities emerge or as the market shifts, it is strategic—it impacts responsiveness and value creation. Change is not a value in itself; its value is in value creation, in responsiveness to need, as our market evolves and expands in new directions. Which sheds light on the bigger distinguishing characteristic of architectural decisions—they are strategic. They have high impact. They determine what value we offer and how, and how we will contribute value to, and play a role in, the ecosystem, especially when we take a longer view. They impact system identity and integrity, and set and shape direction.

Also, architectural design is how we address demanding system challenges—enabling capabilities and system properties given forces and constraints impinging on the system. Challenges that require design
attention. They impact structural and organizational integrity and resilience. They are make or break. Significant decisions that need, as Dawn Ahukanna puts it, deliberate deliberation.

The No Decision Decision

Questions about whether design is necessary or affordable are quite beside the point: design is inevitable. The alternative to good design is bad design, not no design at all.
— Douglas Martin

For these decisions, the no decision decision is a bad decision. Or, as Douglas Martin put it: “The alternative to good design is bad design, not no design at all.” The question is not “does the system have an architecture?” but rather “how good is it?”

[Douglas Martin quote via Kenny Baas-Schwegler.]

"Prioritize design, and real agility will be your reward" — Stuart Halloway

Decisions Constrain

‘Limiting or closing off alternatives is the most common understanding of the term “constraint.”’

“Constraints alter the probability distribution of the available alternatives. They make a system diverge from chance, randomness, or equiprobability”
— Alicia Juarrero

Photo by Will Evans, LeanUX 2015

To be sure, decisions constrain — they eliminate options. Alicia Juarrero observes that this is what we commonly mean by constraint — this limiting or closing off of alternatives; this altering of the probability
distribution of available alternatives. But! In so doing, Alicia notes, they make the system "diverge from chance, from randomness."

**Constraints Enable**

But if all constraints restricted a thing's degrees of freedom in this way, organisms (whether phylogenetically or developmentally) would progressively do less and less.'

— Alicia Juarrero

Constraints close off avenues, restrict the degrees of freedom, but if this was all they did, systems, including organisms, would just do less and less, as they became more constrained (Alicia Juarrero).

**Constraints Enable**

"constraints not only reduce the alternatives — they also create alternatives. Constraints, that is, can also create properties which a component exhibits in virtue of its embeddedness in a system, properties it would otherwise not have."

— Alicia Juarrero

"Causality as Constraint"

Instead, "constraints not only reduce alternatives — they also create alternatives." If we take (Alicia's example of) language, the constraints of syntax allow meaning to emerge.
Architecture decisions, at any point along our system's development path but especially early, "put ground under the feet" (Dana Bredemeyer) and enable us to "gain traction." They move us from potentiality to actuality—closing off options but enabling value to be created; to be able to move forward, we have to start to shape the space. To create a bit of clearing in the "great blooming, buzzing confusion" (William James) or fog of uncertainty; to start to do something, and see how we're doing.

We build on these decisions, and they tend to become embedded within assumptions about what the system is, and does, as well as how it is built, we need to probe and test and improve these design ideas as quickly and cheaply as we can, while they are still reversible.

Now, if we return to our framing question, namely "what is missing if all we have is code?" I have obviously primed the answer: architecture decisions. These are decisions that shape, that address cross-cutting concerns, that have non-localized impact on and so aren't typically visible just in terms of code. They are consequential, success and failure defining, constraining and enabling, and potentially, and over time tend to be increasingly, irreversible. Decisions we need to attend to—in making, and communicating, effectively.
Various architecture decision templates have been published, including by Jeff Tyree and Art Akerman then at Capital One (in IEEE Software, so this template and discussion gained exposure and influence), and Olaf Zimmerman at IBM. But Michael Nygard’s simplified (yet well-described) template caught on as a just enough version for documenting architecture decisions in an Agile context. The Architecture Decision Record, then, documents decisions in terms of the statement of the decision, the outcome sought and the forces weighed in the making of the decision, along with consequences or implications of the decision. (The Tyree/Akerman and Zimmerman versions also keep track of alternatives considered but ruled out, and this is valuable too.)

Now, it is tempting to think, given that it requires our expertise, that the decision is where our focal contribution lies. The decision — what we’re going to do, the approach we’re going to take, is the hard stuff. And it is important — if it’s architecturally significant, it’s make or break. Or we wouldn’t make the decision part of the architecture decision set, persisted in the Architecture Decision Record. The higher the cost and consequence of failure, the better we have to understand the problem, including the forces and tolerances.
As architects, we're seeking to surface what tradeoffs we're making. In order to achieve the outcome we're targeting with the approach we're considering, what forces come into play, what are the downsides, what negative consequences do we need to seek to mitigate?

In this example, it is being pointed out that microservices help us achieve scalability and fault tolerance, but come at the cost of additional complexity in designing and managing distributed systems. Other gains include greater development team independence, which we see again in the next example.

Let's spend a moment and read this (above) from Mattias Peter Johansson on Quora, about Spotify (written in 2017).
We see that allowing duplicate instances of different versions of various libraries enabled Spotify squads (teams) considerable independence, removing the need to coordinate with other squads on libraries and versions. Because song size so dominates considerations that it generally falls beneath the threshold of sensitivity for the user, the tradeoff of team freedom for app size is easily (in their view) within the design acceptance envelope.

So in this case, a technical decision is being made for organizational gain (lowering team coordination costs and increasing team's degrees of freedom) at the expense of app size, which works as long as it's below the app user's tolerance threshold for resource consumption.

That Spotify example illustrates this point that Diana Montalion makes: "the value of every decision depends on the context in which we make it." Diana illustrates with The Lord of the Rings and Frodo's journey to destroy the ring: his quest is meaningful within that context.
Sarah Mei makes a related, important point: system quality is a joint property of the system and the context. Emergent properties, or quality attributes of the system, aren't in this or that line, or lines, of code—we don't see them, when we look at lines of code. They emerge from interrelationships within the system, and with the environment or context of use, for various contexts of use, as well as context of operation, and/or development context(s).

The choices we're making are technical, but the impacts don't remain neatly in the technical space. The tradeoff space isn't just about qualities that impact developer experience, or security properties or operational complexity, but user experience and partner experience through properties of the system in use. And more. So we investigate the upsides and downsides of our technical decisions, in these various contexts.

[Aside: I had a version of Cook and Rasmussen's Dynamic Safety Model at this point, but it was one of the things I eliminated during practice runs of the talk, to bring it within 90 minutes. It can be used to inform how we think about technical debt, for example — in that case, we're dealing (in this view or modeling frame) with developer workload boundary and we can think of the "performance boundary" as the changeability or
adaptability boundary (the failure threshold has to do with coupling and other forms of debt, that thwart adding features and raise cost of change). In the operations case, the performance boundary has to do with scalability and resilience — failures that cause loss and/or distress for customers and news vans to line up outside headquarters.... Etc. ]

"We're trying to find habitable zones in a large multidimensional space, in which we're forced to make regrettable, but necessary, trade-offs." — Robert Smallshire

The design envelope is multidimensional and we need to figure out what the dimensions of sensitivity are, what the thresholds are beyond which we fail, but which we may not even notice until we hit them.

![Decisions: Tradeoffs](image)

That's a lot to bear in mind. Literally in mind. We all have mental models. They are all imperfect. And they are all different. But they are the basis for our decisions — consequential decisions; decisions we may not be able to back out of, especially as other decisions become layered upon, and even entangled with, them.
So we have this composite structure for decisions, where the decision needs to be articulated in terms of the context, of the outcomes sought and the forces at play.

The contribution we make, is not just a good decision, but a decision that is good for its context.

And making decisions, as well as conveying decisions, is as much about what is relevant in the context as it is about the decision we make in response. If we want to make better decisions, and convey them well,
To make a decision, we need to have a (good enough) conception of
- Desired outcome(s)
- Forces and constraints
- Consequences and side-effects

Where these goals and forces arise in various contexts — development and operations, the contexts of use, and the broader value network. We're concerned with factors impacting developer experience, and forces arising from development constraints, capabilities, organizational forces like coordination mechanisms and costs (differing for collocated and distributed teams, say) and taking into account trends in the technology ecosystem. And so forth, also for operations engineers, management teams, security teams, as well as customers and users in various segments and at different points on the user path, as well as partners in the value network — channel partners, others adding value to our products, and more. Architects play a unique boundary spanning role, and we need to see (architecturally significant) impacts others would miss (because their locus of responsibility and experience is more focused).

Of course, this is a lot. These contexts are all important, and to avoid being overwhelmed, the "as relevant" is
there to remind us that we have to use judgment in determining what to pay attention to. What do we need to be paying attention to, now? And what do we need to table, but be mindful about returning to, later? And what is tantalizingly almost relevant, that is hovering in peripheral vision, that we might, at some point need to turn our attention to?

[Aside on the heart and circle: the red heart is the architecture at the center of what we (as architects) do; the blue circle is context; this points forwards and back to color in the slide titles and provides a quick indicator of what the slide is about]

Horst Rittel, the design theorist who coined the term "wicked problem," also said "formulating the problem is the problem." Donald Schon, another pioneering system thinker, wrote (in *The Reflective Practitioner*): “problems do not present themselves to practitioners as givens. They must be constructed from the materials of problematic situations which are puzzling, troubling, and uncertain.” Design is fractal and emergent. We move (in both directions) between problem and solution,
constructing a theory of the problem and theory of the solution (Peter Naur).

"what has to be built by the programmer is a theory of how certain affairs of the world will be handled by, or supported by, a computer program." — Peter Naur

It is a dialogic process, where interactions among the team, and with the system and users, inform and make more shared, the theories of what the system is and why it is that way, and how the code addresses what users need, and what its designer-developers need. But the team changes. The context changes—the system. Changes. What to do?

In a book written for building architects, but which has translatable lessons for software architects, Eliel Saarinen is quoted: "Always design a thing by considering it in its next larger context." Design a chair to be used in conferences in that context, which differs from your office and designing a chair for those demands and that setting.
So we design the parts of our system in the context of the system, the system in its various contexts—of use, operation and development. And that system-of-systems in the context of the ecosystem and its various value networks. Recognizing that the system changes its contexts, means recognizing we're designing the system-in-context — not just the system, but the context too. (While we have limited degrees of design freedom with respect to the context, everything the system takes on, impacts its (various) context(s), so we are redesigning at least some aspects of the containing socio-technical systems and broader context.)

Alternately put, to develop our "theory of the problem," or to "load" the context into our mental models, so that we can uncover this multidimensional tradeoff space, we need to ask (not just) "what do users need?" but also "what do developers and testing need?" and "what do our operations and security teams need?" and "what do others in the value network need?"

We architect across — across boundaries: across not just the code and the teams involved, but across the internal system design and design of the system-in-use or system-of-systems design; across the different languages and concerns of these different spaces, the technical language of code and test and integration,
deployment and operation, and the languages of the domains where the system is used; across the turfs and sense of ownership and decision responsibility; across views and perspectives; etc.

But of course, we can't attend to everything, at least not all at the same time, in detail. We "zoom out," as it were, to scan the ecosystem or value landscape, to identify opportunities and challenges that do warrant closer attention. To set framing for the problem, to understand the trends and forces that shape and constrain it. To get a bearing on the ecosystems that are or will be impacted.

One way to get many perspectives represented in a (literal) "big picture" of the context, is to use the Context Map approach developed by David Sibbet at The Grove. To get a sense of how much comes together in a map like this, we're going to develop a Context Map or Competitive Landscape Map, from the ground up, as it were.
So we start with a green line on the horizon. And the idea that there is an opportunity open to us, that is worth exploring. So here's the thing. We have these disasters like the fires in California and the hurricanes and flooding on the Gulf and East coasts, and other disasters. And while I'm commuting or cooking dinner and listening to the radio, I might get to hear one person's story, or some person with a lot of followers may retweet someone's GoFundMe. We know that others need help too, but we don't have access to their stories, we don't know who they are or what they need. And even when it comes to the stories that are told, we don't know what the response was — did it fill the need?

Now, we have our horizon. First, we'll add some structure. Or structures..

We're looking for who would use the system, and what their concerns are. In this case, people who need help, and people who want to help, are central, so we make space for them, and start to contribute what we know or sense their concerns, goals, frustrations, fears, are. So those impacted by a disaster want to rebuild
their life, but are concerned about dignity and privacy. Those who want to help, want to figure out how to help in a way that makes an impact, a real difference, for someone. Helps recover and rebuild.

Of course, there are other players who contribute in this space, and we don't want to duplicate what they already do well. They're part of the landscape we seek to understand at least enough to differentiate and partner. There are the early responders to large-scale disasters, like FEMA and Red Cross. They mobilize resources at scale. There are local organizations, like shelters, food pantries and Goodwill and Habitat for Humanity. These may be impacted by the disaster themselves, or overwhelmed following a larger scale disaster. The Human Utility (currently serving people in Detroit and Baltimore) deserves special mention—helping get overdue water bills paid, to help people with essential life needs. There are the likes of GoFundMe, that together with social media to get the word out, can be used to raise money, but these requests can be drowned out on general-purpose social media.

We add "winds of change" or trends in the market and technology space: increasing number and impact of
climate change disasters, prevalence of mobile and social media, but with it increasing qualms about privacy. We identify economic and other forces, like state and government regulations, at play. We identify challenges: spikes as media drives attention to a disaster, but rapid drop-off in attention thereafter; concerns about how to fund the system; and so on.

As we fill out more of the landscape, the "big picture" we're all helping to fill out, becomes more "our shared picture." Something we've heard different perspectives on, given different stakeholders in the room. And out of all of this, we're starting to glimpse a system worth building. And we're learning from what others in the room bring, in terms of experience and insight, but also their assumptions and goals or needs and positions.

But. Why architects? We have a contribution to make, and understanding to take, from being involved.

"When we take on architecture responsibilities, our center of gravity shifts from technical to delivering business value through technology." — Dana Bredemeyer

We're technical leaders, and

"Leading is helping people to make sense of the world." — Esko Kilpi

Rich pictures allow us to informally and quickly model socio-technical systems as (collaborative) interactions among people and systems, and systems-of-systems. We can "sketchprototype" to explore various ways we might go about the need we sense, to better understand it, and how we might go about addressing it. Asking (and exploring visually, just enough) "how do things work now?" And "how might they work?"—what would produce "preferred situations" (to refer back to Hebert Simon's characterization of design)?

Here, we've come up with the idea that we could support the creation of caring circles, where someone becomes a caring circle coordinator and sets up a caring circle, enrolling friends, or colleagues or some group, maybe their reading circle or friends in the local Papers We Love community, to help an individual, family, or organization affected by the disaster, that needs help. And we use rich pictures to explore how this might work. We can try out different ways this may play out, in different settings, and it's all as cheap as a sketch, and getting a few people to sketch-storm together on what they'd like to see.
So we took in the landscape, at a high level but sufficient to see across the ecosystem(s) we expect to provide value within. Then we narrowed our focus just enough to explore the system, and how things work now, and how the system might play a role, within those systems of interacting systems. And now we focus on individuals, but we take care to include within our compass not just direct users (or would-be users) of the system, but others who are or would be impacted. Developers, testing and ops engineers, senior managers. Others in the value system who we want to partner with, or provide value to, who we don't want to disrupt but rather further enable. And so forth. But importantly users*. The various design tools need to be in our kit to use just enough, just in time, including interviews and shadowing. Still, Empathy Maps (from *Gamestorming*) are a useful way to imaginatively enter the experience of these various stakeholders.

[* We don't want to build technology-centered systems, though we can be pulled in that direction — “Systems develop goals of their own the instant they come into being” (John Gall, *Systemantics*). Further, we want to scan for environmental impacts, paying attention to negative externalities...]

And Stakeholder Profiles can be a useful way to capture and summarize the goals (business/personal and...
system) of the various stakeholders, so that we keep in view the desired outcomes across the space that we need to weigh and balance as we make the tradeoffs that result in the decisions we make, rather than alternatives we decide not to pursue. It serves to understand the space, and then to remind ourselves and others, that architecture decisions require system perspective, weighing, resolving or balancing forces that arise from, and impinge upon, various stakeholders and systems.

So we iterate, at broadest scope, informing the strategic direction for our system development, and at more narrow scope, informing and probing and improve-adapting the design work, as it unfolds.

At ecosystem scope (given the system or system of systems we're designing), we're exploring value and relationships, to determine how we will differentiate, creating and delivering unique and compelling value. We're looking at leverage points and what else we need to do, strategically, to be sustainable. (For example, iTunes was essential to the success of the iPod, so digital rights management factored. For Tesla to be successful, a network of EV charging stations was needed. The key is looking for what else is needed, to make the value we're trying to add viable.) And we're investigating opportunities to partner with others in the ecosystem, to create a value network.

A Business Model Canvas is another "big picture" view, that pulls together how the business creates value for customers (value propositions delivered through channels to customers segments, activities and resources needed to create value and partners in value creation) and for the business (revenue streams and cost structure). While a Wardley Map shows a value chain where each component of value is shown on the evolutionary path of its ecosystem, in order to make strategic decisions about where to "play" in the value network.
A Customer Journey Map explores user experiences over time as they come in contact with different parts of the system (or system of systems, as in the case of a patient going through treatment in a hospital, interacting with different departments and services, and people/roles). This informs UX design (and information architecture), as well as capability design. EventStorming and DDD’s domain models also have this quality of dual use — they help understand the domain/problem, but in a way that directly informs the design of how the system will be built (design of the system internals).

At any rate, we can do this in the early stages, as we explore how the system will work, in terms of interactions over time. Building, elaborating, nuancing our sense of how things work, and how the system will fit into and support or enable that. And what our system needs to be and do.

And we can return to these tools, as judgment determines, to explore context shifts, to explore how to serve new customer segments, and so forth.

Maps, maps and more maps

- *Impact maps:* goal, who can impact goal (+/-) and how can they obstruct/help, what can we deliver to support impact/goal
- *Assumptions maps:* desirable: do they want this; feasible: can we do this; viable: should we do this
- *User story maps*

We explore assumptions and relationships between outcomes or goals and what various players might do to
achieve or impede them.

The idea isn't, by any means, to use all of these tools. It's useful to notice that different views draw out, or into view, different facets of "the problem." We do need to work some—just enough—at different "zoom levels" and switch our point of view. Work some at the ecosystem level, to set direction. Work some at the system level, and take into account various stakeholders including, but not only various users of the system. Just enough of this, to figure out what's most important to start with. And then we iterate and probe or test our assumptions, suppositions, and working theory of where value lies and how to build it. In sketches and models, just enough, where that is good enough, and in code, to deliver and assess and evolve the system.

What’s Going on Here??

Product Owners and User Experience
Designers, step aside — Ruth's bringing in the architects!

Not that!

Partner well (but just enough), to bring technology capabilities to design table, and to build up "theory of the problem" to inform design choices/decisions

Ideally, anyone on the team can be involved if they want to. Usually this is seen as the purview of Product Managers and Product Owners, and (at least lead) UX designers. The idea here is for software architects to partner well with product owners and user experience designers, to bring technology perspective to the design table, and to build mental models that will inform tradeoffs and decision making. It is about building just enough insight into the context (what we sometimes call the "problem space" but this is a rather unfortunate characterization; and yes, I do it too).

At any rate, we (architects, whatever our title) bring technology know-how/why/what/when/what-if and what else/what then to the strategy and design table. We can help answer: what's on the horizon; what's in the landscape; what does it mean in terms of how we can differentiate; can we build this capability, and should we? And by bringing developer and ops engineer experience explicitly into the design space together with user experience, we're making trade-offs that contend explicitly with the various outcomes and forces that shape sustainability in all these ecosystems. Sustainability in all the senses—technical, economic, social and environmental.
But as much as being involved in this facet of design is about "loading" the context into our mental models to inform decisions that inherently involve tradeoffs, it is primarily about designing the system. Deciding what capabilities are identity-defining and critical, and what we should build now and soon, to lay foundations technically and in terms of value users and others in the value system care about, and want us to build on, creating pull for the design effort. Use Case Diagrams (top level, and drill-down views) show system capabilities we have decided to build (and external systems, for capabilities we will leverage or collaborate to offer). And Use Case Diagrams are useful as we add capabilities, for we can see where capabilities are not yet supported and new capabilities must be added, and where capabilities need to be modified or extended.

Use Case Diagrams may seem passé; they're a thing we tried but is now in the rear view mirror. I still like them for their "system-on-a-page" view, that allows drill-down into the high-level capabilities that matches our ability to drill-down in the components view (when we look inside the system, at its internal make-up of components and relationships). Capability Maps (showing capabilities and relationships) and User Story Maps can play the same role. The important thing is to have a way to represent the capabilities, to assess whether the system will do what is needed, to be fit for purpose. And then to take those capabilities and transform them into responsibilities of the system (informing architecture views).
Simon Brown's Context Diagram (the first C in Simon's C4 architectural modeling approach), gives us the same ability to summarize on a page the users and external systems, and high-level identity-defining and essential system capabilities. The drill-down in C4 from the Context Diagram is into the system, to the Containers (second C) view, and that is good too.

(I chose Simon's Tech Tribes example for illustration here, because it has some overlap with the Caring Circles system in that we have users who create tribes in the one, and circles in the other; both have content assembly and viewing, and so forth. And there's much divergence, obviously. Differences inform too.)

Taking stock of where we are then: Architectural design is system design. And system design is contextual design — it is inherently about boundaries (what’s in, what’s out, what spans, what moves between).

It reshapes what is outside, just as it shapes what is inside.

Taking stock of where we are then: Architectural design is system design. And system design is contextual design. One way this expresses itself, is in decisions about what the system is and does, and how that reshapes what its containing or collaborating (socio-technical) systems are responsible for, and do, and how. That is, design is not just about design of the system (and its internals), but its impact on other systems. Of course there are limits to what we can do, and hopefully we aren't redesigning users so much as offering them opportunities to redesign or redistribute their activities for themselves—but the point is to not ignore
these effects, nor the forces they produce. And further, design of the system internals, needs to inform and be informed by this design of the system in context. The capabilities of the system, and the system properties.

"We shape our tools, and thereafter our tools shape us." — Marshall McLuhan

"The iron law of tools—that which does for you also does to you" — Gene Hughson

So we've discussed using visual models to explore value, and hence system capabilities. But system design is also concerned with the properties, or characteristics of the system. In this roughly 10 minute (starting at 1:12) talk, Russ Ackoff covers and illustrates the key characteristics of systems. Notably, a system has properties that none of its parts have, on their own. When we take a system, decompose it into its parts, optimize the parts, and put them back together, we don't even necessarily get a working system. To see this, imagine you have the best automotive engineers in the world pick the best carburetor, the best fuel pump, distributor, and so on. Now ask them to assemble those best parts into a car. Most likely they can't because the parts don't fit, but even if they do, in all likelihood it doesn't work very well. And at any rate, we can't say anything about the properties, since they are emergent from interactions among the parts, and with the context (stopping on gravel versus pavement, etc.).
To design for more the properties we want, we need to get a sense of what we want. For design to be intentional (towards preferred), we need to bring what we intend more into view.

A lightweight way to do that, was developed by Michael Keeling based on the work of the Software Engineering Institute (SEI) on system quality attributes and quality attribute scenarios. It has the properties of many of these visual meeting/workshop style whiteboard/flipchart tools, in that is supports a group coming together to identify which characteristics or quality attributes of the system need to be prioritized (axes, and placement on the axis, of the qualities kiviat) and refined as attributes of the system that can be assessed.
Rebecca Wirfs-Brock introduced another variant, leveraging idea from Tom Gilb, which emphasizes that there is an acceptance range, not just a target. She likens this to landing zones on an aircraft carrier. At any rate, there is a minimum level (for qualities where more is better), below which we fail, and an outstanding level where we would delight users (and, potentially, customers, if these are distinct) but where the cost of reaching that level is such that our target is set lower.

Quality attributes shape our (multi-dimensional) design envelope, and having a range rather than just a specific target recognizes that we're making trade-offs to balance across various properties or qualities, and we may have to accept less than outstanding, or even the target, to be adequate along another dimension.

The team at the SEI, including Rick Kazman, Len Bass and Paul Clements, has done a lot of good work on quality attributes and articulating them unambiguously, so that we can evaluate our decisions, designs and systems for how well they do against them. If you like the idea of incrementing towards better fit to fitness functions, getting a clearer sense of target properties is important. And there is a good body of work behind quality attributes that is worth drawing upon.
Decisions are about tradeoffs, but the design envelope is multidimensional (alternately put, the fitness landscape is multivariate) and we need to figure out what the dimensions of sensitivity are — what the thresholds are beyond which we fail, though we may not even notice them until we hit them.

We seek to create a system that is fit to purpose and to context, for its various contexts of use, and contexts of development and operation. It takes weighing and resolving forces (where the forces are coming from every whichwhere), balancing and making tradeoffs. Constraints may be bounds (finite amount, physics determining bounds on speed, etc.) in technology, physical limitations, or organizational and economic. To cope, we try to stay aware of what is shaping, determining success (how we create value, and for whom, because it's not just about keeping customers engaged) and failure. We're dealing with complex, interacting matters of integrity — structural integrity and robustness and resilience (adaptive capacity not just recovery capacity); organizational integrity and ethics; design integrity and cohesion, fit, and adapted affordances. And matters of sustainability — how do we ensure that what we do now, doesn't undermine our future — in each of the ecosystems we're embedded in and impact, from the technology space (where it surfaces as tech debt, and tech entropy) to the environment. It's not just that business goals, goals of different groups within the organization, and individual goals, and customer (and user) goals rarely line up so tradeoffs have to be made, but further, none of these may understand their goals well and more, they change, as contexts and needs and attention shifts. And then there's the question of limited resources and constraints on communication and coordination. We have to make it tractable.

"The better you understand the problem, the closer you can design to tolerances." — Dana Bredemeyer

"A good Service Level Objective is just at the point the user says "meh" and presses refresh." — Liz Fong-Jones
We need to use judgment, to steer our discovery process, directing design attention where we need to, given the moment we’re at. Disciplines like "You Ain’t Gonna Need It" (YAGNI) and deferring to the "Last Responsible Moment" need to be held a little more loosely, as we decide what it architecturally significant and needs to be explored, probed, better understood and assessed because it changes what’s ahead. Maybe it sets us up to be more responsive and agile. Maybe it shapes system or organizational identity, or is key to system integrity. Maybe it’s critical to what kind of system this is. Each "maybe" signifies a judgment call, or set of them. There’s the joke: "what answer distinguishes the architect?" "It depends." But a good architect tells you what it depends on.

So we look for defining moments, telling, shaping moments, and seek to identify what matters. In part, this is about "lifting our head" from the work, and identifying challenges ahead, and deciding how to explore and discover what is significant, and how to explore what our design options are. Writing code. But with an eye, sometimes, on the horizon, scanning for what’s ahead, what’s taking shape.

Agility is about sensing and responding—observe, orient, decide, act in quick cycles. We need to ask: what,
at this extraordinary moment, do we need to be paying attention to, that no-one else is paying attention to, but which is significant. Recall, architecture is about what is make or break. What is setting up to cause us to fail, and how do we not do that? What opportunities do we have to add value, and how do we do that? What should we do now, and soon, and what can we defer. We're observing and orienting at different scopes — tactical and near-term, sometimes, but staying alert, and maintaining a discipline of scanning more broadly, across boundaries, across time, from time to time. As we accept the adaptive responsive nature of Agile, and co-evolutionary development, we let go of the need to be done. Now, strategy, design and coding-testing, all become ongoing, but rely on a more dynamic process of observing and orienting, to decide and act. And probing-testing the actions we take, to reverse out of them before they become "baked in" to assumptions and habits (our own, or users and partners in the value network), or workflows, or coupled into more structural entanglements and dependencies.

Now, it's all very well talking about visual design in the domain we associated with, well, designers. When we're trying to understand the world of systems and people our software will fit in, and shape and be shaped by, it's bigger—so much bigger!—than our code and system. So models (representations, abstractions to skim for relevance and to understand, etc.), developing theories of "the problem," and experiments to test them, make sense. Including and reiterating for emphasis, the models or sketches part.

Up to this point, I've been arguing for the involvement of architects in design of what the system will be (the "problem" or "requirements" space). To help build understanding of the domain and the opportunities technology brings to it, to inform the technical design work, and to keep that understanding from becoming out of date, by remaining plugged in to design of system capabilities as the domain and the system evolves. And we need to design across what the system will be and how it will be built, because tradeoffs don't stay neatly compartmentalized. So there's that.

As we transition to talking about visual design and its role in designing how the code will be structured and how the system will be built to yield those capabilities and properties, let's take a step back and consider what we get from making design visual.
Drawing on the walls is not new—we've been doing it since at least 33,000 years ago! If you haven't seen *Cave of Forgotten Dreams*, about the Chauvet Caves in France, you're in for a treat. Only discovered relatively recently, and, with few exceptions, closed to view, it is a striking and touching connection with amazing artists living so long ago. And it speaks to the value of images in conveying across time, but also the limitations. While the paintings marvelously capture animals including species not recorded in other cave paintings, they don't tell us much about human life at the time. We make guesses about the uses of the paintings, but they're guesses.

Jump forward tens of millennia, and we have engineers drawing on walls. This scene with (Taraji P. Henson playing) Katherine Johnson in *Hidden Figures*, wonderfully captures why. We draw "out loud" (so to speak) to help ourselves reason, to help others follow our reasoning or to explain it, and to collaborate—to create a "shared thoughtspace" where others can understand our idea and the reasoning that supports it, add to it, get ideas of their own, and so on. "I see what you mean" is literal. And in seeing, and hearing the thinking that also helps illuminate what is being drawn, we can question our assumptions, catch flaws, share different perspectives and alternatives we see, and more. Fundamentally, we're sharing, teaching/learning, and/or
collaborating in a dynamic live-interaction way, so we can respond to questions, and ideas or concerns, capture more of the picture, the forces in play (including the "politics" and elephants and hippos in the room), etc.

**Visual: Drawing in notebooks**

Leonardo Da Vinci’s notebooks:
used sketches to
- observe (more attentively)
- study, think, reason, to puzzle things out
- understand

Engineering notebooks have an inspiring history too. I used to think of Leonardo da Vinci as an artist-inventor, and it wasn't until I read *Engineering and the Mind's Eye* that I realized how much of a scientist and engineer he was—recognized as such in his day, too, holding official positions as engineer including "painter and engineer to the Duke", and “First painter, architect, and engineer to the King."

Da Vinci was intrigued by problems like inertia, friction and resistance, and investigated mechanisms and mechanical elements such as screws and gears, pulleys and weights, with a piqued visual curiosity. One of the things that stands out for me, in Da Vinci's method of studying anything from physics to anatomy, is that he sought to understand not just structure but dynamics, and the interaction of structure and dynamics, or how things work and the principles that govern how they work. And he used his illustrated notebooks to document and pursue his study. It is clear he used the pen to see, to observe more closely, and more closely still, to probe and explore structure, and the principles of dynamics that relate structure, constraints and behaviors. And more. His notebooks served not just to record, but to teach—himself even. Drawing and note taking was an instrument of learning about what he was probing with his eye, his mind, his mind's eye, and what he externalized on the page, pages, volumes.

"A recent and exhaustive analysis of Leonardo as a scientist by Fritjof Capra argues that Leonardo was a fundamentally different kind of scientist from Galileo, Newton, and other scientists who followed him, his theorizing and hypothesizing integrating the arts and particularly painting. Capra sees Leonardo's unique integrated, holistic views of science as making him a forerunner of modern systems theory and complexity schools of thought" — wikipedia
Da Vinci advanced various fields of science and engineering, figuring out, and prefiguring advances that we often ascribe to those who came after him: he foreshadowed Copernicus by 40 years, declaring "IL SOLE NO SI MUOVE" ("The sun does not move.") And adding, "The earth is not in the center of the circle of the sun, nor in the center of the universe." Further, 200 years before Newton, he wrote "Every weight tends to fall towards the center by the shortest possible way." He was likewise prescient in other fields. 400 years before Darwin, he placed man in the same broad category as apes. (Gimpao Ni Ei Suu h)

Da Vinci's notebooks capture and extend understanding of phenomenon of nature and machine, and include numerous inventions, some built in his time, others later—such as the lens grinding machine pictured. These inventions, we might note, were constructed as sketch-prototypes so compellingly drawn as to both persuade feasibility in many cases, and to inform construction. (The Archimedes steam cannon is a fun story).

Though his scientific notes are articulate and precise, for da Vinci, illustration took precedence—the drawing does not illustrate the text; rather, the text serves the illustration, only adding detail best expressed in words. To this end, he developed principles of graphic representation — stylization, patterns, and diagrams.
As important as sketches, visual models and annotated diagrams are to thinking alone and collaboratively, and over time, a lot can be done in text. Take the architecture of the US government—its key principles, structures, and mechanisms, are described in the US Constitution. It is marvelously minimalist at just 4 pages.

It is expanded on in the Federalist Papers, which are reasoned arguments for different aspect of the Constitution. They take a technical white paper form, describing the context and problem, the solution approach and how it addresses driving concerns, and implications.
And in Episodes, Ward Cunningham advocated writing short, to-the-point, technical memoranda to address key subjects not expressed in the code, that need contemplation and reflection, and involve enough detail that they are not easy to recall.

Still, the power of adding sketches or visual models is evident in Randall Munroe's book, *Thing Explainer*. Randall set himself the constraint of using only the 1,000 most commonly used words, and the book works astonishingly well. (Note, the words constraint was for fun, not to be condescending to kids.) Partly, it works so well due to Randall's grasp of these various systems, and is a function of strong visuals, along with the annotations. I recommend *Thing Explainer* to architects a lot, because that's what we need to see ourselves, in good part, doing — understanding structures and mechanisms and explaining them. And doing so with the aid of sketches, where we can make the concepts and dynamics visual. More on that soon.

For now, let's just underscore: as we train ourselves as architects, we'd do well to be taking Leonardo da Vinci and Randall Munroe as inspiration, using drawings illustrated by text to discover and teach the design principles that underlie our complex systems — for our own systems, and others. Study design, through the
pencil. And communicate design, through the marker. And other diagramming or modeling tools, as and when we want to convey something more polished and thought out.

Diagrams and text serve importantly to communicate and learn, but I want to add to Simon Brown's point that we use diagrams or visual models in design to do design. To create better designs, and to express our designs. True, this helps in communicating our designs and sharing what we've learned, and have to convey.
We're learning about embodied cognition (involving more than the brain) and extended cognition where we use the world to think — not just externalizing memory to books, notes, the internet, but also using the world to think and reason. Like using an abacus to do calculations. And it turns out even spiders appear to use their webs as devices to figure out what to do (next). For example, extending their web in the area where bugs have been caught.

So here's a problem to illustrate. Imagine a monk starts up a mountain at 8am and arrives at the top at 8pm, and the next day returns by the same path, starting at 8am and getting to the bottom at 8pm. Can we say with confidence that he was at the same point on the path at the same time of day, on the two days? How do we know?
One way to think, is to draw, and the diagram illustrates that we can say yes. Another way to think about it is each of my hands is the monk on the two days, and one hand will move along the path in one direction, and the other hand is the monk starting at the other end of the path, and moving in the other direction on the same path. My hands have to meet at some point, at the same time. As important as the illustrations are to the point that we can put something in the world to help us think, it's also illuminating that some people will still not see it, and these people are important too. We can try to illuminate the solution different ways, but our perspectives differ, we're looking for a catch, and trust and credibility may factor, etc.

Designing is thinking. Hard! Reasoning, relating, making trade-offs. Architectural design is thinking across the system — that's a lot to hold in mind. Visual thinking expands our mental capacity, and sketching these images and models expands our processing power still more. The mind’s eye is resourceful, but it can’t readily see what it’s missing. Sketching creates external memory, so we can hold more information and see relationships, make connections, spot inconsistencies and gaps, and draw inferences about causalities and consequences, highlighting what we need to exercise or test further.

Sure, code externalizes thought too. And code (and TDD) is an effective design medium. Still, models help us think through aspects like overall structure, or particular design challenges, even before we have (all the) code. We play out ideas in our imagination and on paper/whiteboard/screen, to define, refine, re-imagine, and redefine the dominant architectural challenges and figure out architectural strategies to address them, identifying where to do more focused experiments in code. This may sound like a lot, but architectural decisions are make or break, structurally or strategically significant, and key to design integrity. They warrant closer attention. Judgment factors.

Drawing views of our system, helps us notice the relationships between structure and dynamics, to reason about relationships that give rise to and boost or damp properties. We pan across views, or out to wider frames, taking in more of the system and what it does where and how and why (again, because we must make tradeoffs we need to weigh value/import and risk/consequences and side effects).
To be agile, that is to be responsive (which includes sensing as well as responding), adaptive and resilient, we need to learn in the quickest, cheapest medium that does the job. With sketches and models, when that allows us to probe and explore, and discover well enough, for the moment we’re at. A sketch prototype is easy to reverse or back out of—at least, it is as easy as letting go of our ideas, which can be hard, I acknowledge. But easier than code we build to experiment, though we do that when models don’t give us enough insight. The key idea is not to return to “big design up front,” detailed out in models, but to use sketches and models as a design aid throughout. Some upfront, and more along the way. It is a tool in our design toolbelt. And we need to be okay reaching for that tool, when it is good enough for the decisions and design work at hand.

Sketches, diagrams, visual models (of facets of the system) have benefits over text in terms of showing relationships and interactions, and leverage what our visual perception is good at — spotting patterns, noticing what’s missing or surprising. We can use placement and color and figure to show, and we can use what is not shown, to express, To enable us to cope with complexity, by abstracting away detail unnecessary to what we are currently reasoning through.

Models help Reason, Explain, Design, Communicate, Act, Predict, and Explore (forming the acronym REDCAPE) from Scott E Page, *The Model Thinker*. (Via Alan Hakimi)

"Maps help us see fuller picture, devise/test strategies" — Brenda Michelson

Using drawing to observe more closely, to notice relationships and patterns, studying systems in terms of structure, dynamics and properties (and tradeoffs among them) to diagnose and improve, to get more the properties we want and/or less of what we don't want.

In sum, making it visual, helps us think — better, together, and over time.

"It's weird to me that visual design and diagram isn't even table stakes for doing design work anymore. It's like forgetting how to run." — Hillel Wayne
When we say that we value “people and interactions,” in part (I think) we’re saying we value collaborative work to get things done, over the “stocks and flows” of documentation and workstages. And in-person collaboration happens in a context — it may be convened around a keyboard, or a whiteboard.

We use drawing together, on the walls (or whiteboards), to convene collaborative design work; it convenes the conversations that inform. Inform the design work, and inform those doing it. As we draw out, we draw in. Working together, unfolding the design, engages minds (and hands, in the embodied cognition sense). Participation persuades. (Which has its dangers, and we have to be aware of our inattentional blindness and other cognitive and group biases, etc.) Sure, we can mob program, and do. The point is, we can mob model too.

We need to capture the design, and the reasoning (tradeoffs, why and wherefores, assumptions and implications, etc.) to hold a record to communicate with our future selves, and new team members, and so forth. But the first value is in the doing. Facilitating the thinking (together) that goes into the design.

Visual design (sketches, diagrams, models, stickies, decision matrices, etc) is a crucible for collaborative work — can be, should be. Now, let's turn to visual design of the system, in the sense of what the system is made of and how it works. Design of the guts of the machine, at least at the architectural level.

We're going to be more illustrative than complete here, but the key idea is to explore how diagrams and models can help us create architectural designs, and express the design both to improve it, and to convey it.
First, to set context, let's level set on some terms. System design is about coherence and integrity. Recall: when we're talking about integrity we mean: structural integrity (robustness and resilience); organizational integrity and ethics; and design integrity and coherence and fit. Fit to context and to purpose; internal fit and conceptual integrity.

**Software architecture** refers to the high level structures of a software system [...] Each structure comprises software elements, relations among them, and properties of both elements and relations.

— wikipedia/Clements et al

We've talked about software architecture as "the significant design decisions that shape a system, where significant is determined by cost of change" (Grady Booch). Another influential characterization comes from the team at the SEI: "Software architecture refers to the high level structures of a software system and the discipline of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of both elements and relations." (Clements et al.)
What is agile? One way to understand or characterize what something *is*, is to characterize what it is *not*. Not agile is unresponsive or unchanging in the face of change. It is rigid, and resistant to changes in direction.

<table>
<thead>
<tr>
<th>Not Agile?</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Rigid: not able to be changed or adapted</td>
</tr>
<tr>
<td>- entangled</td>
</tr>
<tr>
<td>- Inertia: the resistance of the object to any change in its motion, including a change in direction</td>
</tr>
</tbody>
</table>

---

"You reach for the banana, and get the entire gorilla"  
– Michael Stahl

Similarly, to characterize good architecture, we consider what characterizes a not good architecture: a "big ball of mud" springs to mind. When you have a "big ball of mud," as Michael Stahl vividly put it, "you reach for the banana, and get the entire gorilla." Entanglement mires and makes a system hard to change.

"If you think good architecture is expensive, try bad architecture." – Brian Foote and Joseph Yoder
A modular structure reduces cost of change by (and to the extent that it achieves) isolating change, shielding the rest of the system from cascading change. In a modular approach, parts of the system that are unstable, due to uncertainty and experimentation to resolve that uncertainty, can be shielded from other, better understood and more stable parts of the system. Parts can be plugged in, but removed if they don't work out, making for reversibility of decisions that don't pan out. They can be replaced with new or alternative parts, with minimal effect on other parts of the system, enabling responsiveness to emerging requirements or adaptation to different contexts. Further, it's a mechanism to cope with, and hence harness, complexity. Partitioning the system, reduces how much complexity must be dealt with at once, allowing focus within the parts with reduced demand to attend (within the part) to complexity elsewhere in the system. We give a powerful programmatic affordance a handle with minimal understanding to invoke it, and can selectively ignore (not all the time, but for a time), or may never even need to know (it depends), its internals.

In a modular approach, parts of the system that are unstable, due to uncertainty and experimentation to
resolve that uncertainty, can be shielded from other, better understood and more stable parts of the system.

"Uncertainty is not a license to guess. It is a directive to decouple." — Sandi Metz

Parts can be plugged in, but removed if they don't work out, making for reversibility of decisions that don't pan out. They can be replaced with new or alternative parts, with minimal effect on other parts of the system, enabling responsiveness to emerging requirements or adaptation to different contexts.

Further, it's a mechanism to cope with, and hence harness, complexity. Partitioning the system, reduces how much complexity must be dealt with at once, allowing focus within the parts with reduced demand to attend (within the part) to complexity elsewhere in the system. We give a powerful programmatic affordance a handle with minimal understanding to invoke it, and can selectively ignore (not all the time, but for a time), or may never even need to know (it depends), its internals.

“There was a wall. It did not look important. It was built of uncut rocks roughly mortared. An adult could look right over it, and even a child could climb it. Where it crossed the roadway, instead of having a gate it degenerated into mere geometry, a line, an idea of boundary. But the idea was real. It was important. For seven generations there had been nothing in the world more important than that wall.

Like all walls it was ambiguous, two-faced. What was inside it and what was outside it depended upon which side of it you were on.” — Ursula K. Le Guin

“Encapsulation is important, but the reason why it is important is more important. Encapsulation helps us reason about our code.” — Michael Feathers

In talks on software architecture, Grady Booch points [minute 43+] to what he characterizes as “the fundamentals that remain fundamental”:

- clear Separation of Concerns
- crisp and resilient Abstractions
- balanced distribution of Responsibilities
- strive to Simplify — Grady Booch
which I resequenced as shown above, to give us the mnemonic, SCARS* (see The Architect's Clue Bucket, slide 16). Which is appropriate, since SCARS are what we get from experience.

---

**SCARS: Separation of Concerns**

---

"I go along with the natural makeup"...

"when I come to the tricky parts, I slow down"

— Chuang Tzu: "The Dexterous Butcher"

---

Taking the first of the SCARS (Separation of Concerns), here's a story from the Tao. It is the story of the dextrous butcher. In one translation, the master cook tells us:

"It goes according to natural laws, Striking apart large gaps, Moving toward large openings, Following its natural structure.

A good cook goes through a knife in a year, Because he cuts. An average cook goes through a knife in a month, Because he hacks.

I have used this knife for nineteen years. It has butchered thousands of oxen, But the blade is still like it's newly sharpened.

The joints have openings, And the knife's blade has no thickness. Apply this lack of thickness into the openings, And the moving blade swishes through, With room to spare!

That's why after nineteen years, The blade is still like it's newly sharpened.

Nevertheless, every time I come across joints, I see its tricky parts, I pay attention and use caution, My vision concentrates, My movement slows down."

What do we extract, that helps guide us in architecting? Two heuristics jump out:

- follow the natural structure
- when we come to the tricky parts, slow down

This story is used as a teaching story in diverse situations. And yet here it is, yielding heuristics for architecting, which is, after all, at least in part about (de)composing a coherent system. We might challenge the heuristic with: what, in so conceptual a matter as a system formed of thought, written as code, is "natural structure"?

One place to go, in looking for the natural topology, to find the natural shape to follow in creating internal
system boundaries, is the “problem” domain — that is, the domain(s) being served by the system(s) we’re evolve-building. This brings us to bounded contexts and heuristics and guidance in Domain-Driven Design to suggest system and component or microservice boundaries. And to business capabilities in Enterprise Architecture/Business Architecture approaches, that seek to understand the topology, the shape and relatedness, of business capabilities.

Another place to go is back to 1989 and Ward Cunningham and Kent Beck's CRC cards, but repurposed for components, along with a heuristic that takes the form of a neat antimetabole (thanks to Jan van Til's paraphrase of a Tom Graves point): "The responsibility of architecture is the architecture of responsibility." Which points us in the direction of identifying responsibilities related to system capabilities and properties, and the arrangement of responsibilities. Separating responsibilities along the lines of concerns. [This is not the only way we use the term "separation of concerns" in software design, of course.]

We use abstraction in different ways — levels of abstraction, abstractions as elements of code we work with, the activity of abstracting. J. Edsger Dijkstra (The Humble Programmer, 1972) noted:

“The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.”

The abstractions we use to give our system internal form, must, as Michael Feathers points out, be invented. They are conceits. In every sense of the word, perhaps. But they begin as conceptions.
How do we come up with these abstractions, these conceits?

"To find the right abstraction, guess. If it exhibits the right properties, stop." — Jessica Kerr

We're looking for heuristics for identifying architectural elements (or components, or microservices if those are the architectural elements of the system in question). A crisp abstraction has a clear, unifying purpose or cohesive identity; it has, in effect,

a single responsibility at the level of abstraction of the abstraction

This suggests a focus on responsibilities as an avenue to system (de)composition. We identify responsibilities and assign to them components, working in either (and both!) direction: start with a first cut notion of components the system will need, and identify and allocate responsibilities to them; or start with responsibilities evident from what the system needs to do, and factor to find components.
So, back to our CaringCircles system.

Here we have our first guess. We might note that the hand drawn, handwritten nature of it, is a feature not a bug. For hand drawn conveys incomplete, not done, still in-progress and changeable. It is humble, in the best sense. But given my writing, we can't read it, so...
The dual to separation of concerns is a coalescence or cohesion of concerns?

"Things that are cohesive, [...] naturally stick to each other because they are of like kind, or because they fit so well together [...] the pieces all seem to be related, they seem to belong together, and it would feel somewhat unnatural (it would result in tight coupling!) to pull them apart"— Glenn Vanderburg

That is, not only are we looking for what to pull apart, but what to keep together to form crisp and resilient abstractions.

Here we have an artificially simplistic clustering (to illustrate the point), putting things to do with requests in a service, and offers in another. However, users entering and updating requests are Requestors (people asking for help, or their proxies helping them do so), while the users for View requests and Claim requests (i.e., signing up to fulfill them) are Caring circle members. This latter is a “search and make a manual match” feature.
So we update the responsibilities lists accordingly. Further, checking the use case diagram, we notice that fulfillment is missing and add that.

We take a guess as a starting point, and improve on it: Run thought experiments and model (use cases or user stories; focus on one property, then another, etc.) to flush out responsibilities we overlooked in our initial guess.

These lists of responsibilities are a powerful and largely overlooked/under used tool in the architect’s toolbelt. If the responsibilities don’t cohere within an overarching responsibility, or purpose, that should trip the architect’s boundary bleed detectors. We may need to refactor responsibilities, and discover new components in the process. We think that “naming things” is the big problem of software development, but “places to put things” is the proximate corollary.

“Be deliberate and deliberate all the things” — Dawn Ahukanna
As we do this exploration with the aid of models (just as we do when doing design in the medium of code), we're applying heuristics we've developed through experience, and exposure to other people's work (books, and such). Heuristics don't take away the need to think, to reason and try things out. They help us identify what to think about, as we do so, and may suggest how to go about it (better).

In another of the foundational classics of our field, David Parnas (On the Criteria To Be Used in Decomposing Systems into Modules, 1972) proposes:

"that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

That's two heuristics, and thanks to Robert Martin, the second has primary place in the SOLID set of principles that have been a touchstone of software design, and object-oriented design in particular:

Single Responsibility Principle (SRP): a class should have one and only one reason to change

Stuart Brand, writing about pace or shearing layers in building architecture in How Buildings Learn, observed that different parts of a structure change at different rates (with the building site, and then the structure, being least likely to change, given the cost and difficulty of doing so; and the stuff inside being the most likely to change). This observation gives rise to the heuristic:

keep things that change together, together

Thus, one thing we're seeking to do with a modular structure and crisp abstractions, is to isolate change, shielding, as best we can, the rest of the system from cascading change. We can check how we're doing against the heuristic, by running anticipated changes across the architecture, informally mapping a new use case on the architecture, to see if responsibilities that are missing illuminate a different clustering/factoring an resulting abstractions. We're not trying to build anticipatory capabilities we're not sure we'll need. We're simply stress testing the resilience of our abstractions, to see if we've missed a more resilient basis for separating and coalescing responsibilities.

Let's return to Parnas; the other heuristic in his criteria for decomposing (already quoted above) is:

"[begin] with a list of difficult design decisions [...] Each module is then designed to hide such a decision from the others."
Here we return to the TechTribes example and consider in particular the interactions involved in retrieving content from other systems (twitter, github, newsfeeds).

Turning to the Component Diagram, we see components (Twitter Connector, Github Connector and Newsfeed Connector, respectively) serving to protect the core of the system from interactions with the outside.
Which brings to mind Alistair Cockburn's hexagonal architecture pattern. Here, adapters at the system boundary, shield the (core) application (code) from interactions with the outside, keeping business logic uncontaminated by the peculiarities of external agents or systems and their states, interactions and interaction paradigms, and keeping business logic from leaching into user, or other system, interface code. Moreover, ports and adapters are a first mechanism of address for cost of change, partitioning the system boundary into plug-and-play access points.

In the founding classic of system architecture, Eberhardt Rechtin presents heuristics gleaned from his amazing career as a system architect in aerospace, and master teacher of system architects in the seminal program he created at USC. One of these heuristics (is a "turtles all the way down" sort of thing, but applies also at the system level):

"Design things to make their performance as insensitive to the unknown or uncontrollable external influence as practical."

Relatedly, Postel's Law (also called The Robustness Principle) states:

"be conservative in what you do, be liberal in what you accept from others"

I nickname it the "play well with others" principle. At any rate, concerns at the boundary encompass responsibilities to not just plug and play, but to play well.

So we have this interplay between patterns and heuristics and modeling, where patterns suggest design elements and heuristics help guide our attention and design choices, and notice what to be wary of and so forth.
This heuristic (and admonition, really) from Steve Jobs ("Design is not just what it looks and feels like. Design is how it works.") is not just relevant to design across "the skin and the guts" but within the guts, it reminds us to design across structure and behavior.

Design of the system "guts" takes a different kind of attention from design of the system capabilities: we're reasoning about structures (components, interfaces, relationships) and looking for opportunities to reduce dependencies, by exploring options for what we bring together, and what we split apart. That is, we're looking for natural "seams" in the system. We may start with DDD and domain models and bounded contexts. Or with guesses at components arrived at by considering system capabilities (possibly represented as use cases) and responsibilities.

But we're — importantly — reasoning about the interaction among architectural elements to yield capabilities (and related value to stakeholders) with desired properties (impacting user-, operations-, and developer-experience, and impacting system integrity and sustainability), addressing the inherent challenges and tradeoffs that arise as we do so. Which means that we're exploring behavior, to direct discovery as we're
exploring how to build the system, not just its parts. But this impacts how we conceive of and design the parts.

So we're positing structure, and asking "what is this system made of?," but also — and soon — exploring behavior, asking "how will this work?" and "how does this approach contribute to or inhibit the desired system properties (including reversibility and adaptability through modularity and decoupling) and yield needed system behaviors?"

As we do this, we're going to learn more about the responsibilities of the components or architectural elements. And we need to be disciplined about updating the responsibilities and refactoring components/responsibilities to try out alternative structurings, or to improve the current one.

"Structure determines what behaviors are latent in the system." — Donella Meadows

**Iterating Across Structure and Behavior Views**

As we iterate across structure (what, or elements and relationships) and behavior (how it works), we're improving the Conceptual Architecture — we find responsibilities or relationships we missed, we might refactor, etc. [If you're looking to map this discussion to Simon Brown's C4 architecture model, we're improving the Component Model] We're also identifying (possible) interfaces (Logical Architecture).

We generally advocate using just enough UML (or sysML) to get the job done. Where just enough may be UML-ish, if that's good enough. UML is a modeling language, and like other languages, its expressiveness is more than we generally need, support and convey our reasoning. At any rate, we postulate an initial structure, and use sequence diagrams or other behavior diagrams to explore not just components and responsibilities, but to start to identify interfaces.
In this example, we have the use case diagram and a fragment of the conceptual architecture diagram for a reservation system.

And here we're playing the MakeReservation use case over the involved components, and updating the initial interfaces for the components in question. We might notice that with a Communication Diagram, we overlay behavior on the topology of the components, allowing us to keep the topology as a constant across Communication Diagrams, so it's a stable point of reference.
A Sequence Diagram, by contrast, emphasizes timing (well, as indicated by the name, sequence). We find this quite intuitive to create and read. In either case (Sequence or Communication), we're exploring "how it works" — or could work, and if it becomes part of the design, then how it should work. Activity Diagrams can also be used to explore behavior (components are swimlanes, activities reveal responsibilities). If we're designing an event driven system, we'd want to use Statechart Diagrams. The important thing, especially early, is that we can think through interactions that give rise to relationships (and interfaces) and explore alternatives quickly. Looking across these diagrams, we can check that we have captured the responsibilities and relationships indicated by the messages on the Conceptual Architecture Diagram.

We can use these diagrams to illustrate what we mean or intend. Or to show what the system does, to explain it to new team members or those who will use our APIs.

Whether we're using UML diagrams (so that we don't have to invent notation and document syntax and semantics) or ad hoc diagrams (to just go ahead and express the design thinking as it occurs to us to represent it), we're doing the important work of making our individual and team think visible. And exploring design ideas for the overall system structure and dynamics, or key elements such as the design of architectural(ly significant) mechanisms.
While I encourage teams to start with Conceptual Architecture (key abstractions and relationships), we want to soon, and then along the way, capture the constraints of the physical environment in a topology diagram, and investigate implications for our design as we think about distribution and implications of physical boundaries for communication and performance and latency, and other properties being driven by the deployment environment.

“Don’t partition by slicing through regions where high rates of information exchange are required” – Eb Rechtin
Iterating across structure and behavior, or what the system (or an architectural mechanism) is made of and how it works, is a key idea, and it is hard to do with just code as the externalized thinking medium. Alternately put, this is exactly where visuals have their strength, in that they allow us to see, and hence notice what we're missing, relationships and interactions, or states and transitions. So that we can reason about properties and capabilities and develop our "theory of the program" at system design level. It gives us the material to reason more visibly, and sure, we're fallible, but that only means we need to try different means to make our gaps and weaknesses in the design evident as early as we can. Explore alternatives, improve the design. Dive into details where we need to. Write code and build up the system, learning more about the design as we evolve it. But this is a messy process. It is messy so that the end result is less so!

As we draft views of the architecture, and work across them, and as we're building increments and making discoveries, we need to be willing and disciplined about learning even though it means backtracking, and revisiting and revising the architecture. Indeed, in his seminal book on Systems Architecture, Eb Rechtin listed the following among his essential characteristics of system architects:

"High tolerance for ambiguity"

"The willingness to backtrack, to seek multiple solutions"

At any rate, it serves us well to

- develop an architecture decision record;
- document the essential structure of the system, with explanations and rationale (tradeoffs made, connecting dots to desired outcomes, etc.); and
- document key architectural mechanisms (using 2-page spreads such as might be inspired by Thing Explainer (without the simple words constraint) or something like Martin Fowler's explanation of the LMAX architecture. These are nice illustrations of ad hoc visualizations that powerfully help communicate the key design ideas — and the role that textual description plays.

By advocating for visual design, we're by no means suggesting we shouldn't also be disciplined about the conversations and the textual descriptions. We need to make the architecture vivid and compelling. For even though we work on the architecture together, in the team (or part of the team, for large teams), we're capturing the reasoning and design expression for our future selves, and new team members.
Feeding Learning Back into the Design

So we iterate across system views, evolving the architecture. And as we build and deploy more and more of the system, we continue to learn and evolve the design.

Essentially, when we start out, we haven't decided what our elements are, even conceptually. We're just mucking about with ideas for how the system might be organized, toying with ideas for structures and mechanisms, and playing out how different alternatives might work, in as fuzzy and fudgy a way as suits the moment. But we start to tighten down our thinking as we make choices, and as our design thinking matures (meaning we make more decisions and the jell starts to be more viscous) we may start to use more of the modeling power and more support from the tooling.

As we build out system capabilities, it is useful to have support (in tooling like Structure101, Lattix, and Codescene) to show us the shape of the system, and show us "hot spots" of complexity and change, or where the design in the code is departing from the architectural design so we can probe why and if this is a good thing, and so forth. Our tools must not become our masters. But they are useful to see the system, to observe and learn from the design as built — from the perspective of users of the software, and from the perspective of developers of the code, as well as operations engineers. And we should never forget the usefulness of the pencil, or marker, to draw out our "theory of the system," its realized mechanisms, and how they work, and what the key structures are.

And so, as we evolve the system, we're moving between intention, writing code informed by that intention, but doing further design in the medium of code, using tools and modeling to learn from what we did and how the system behaves, and transferring that learning to evolve the design as intent. Keeping the design expressions in pace with the evolving code.

Still, I want to emphasize the learning opportunity we have, when what we have is just design expressions.
We Model: To Test

Models help us try out or test our ideas — in an exploratory way when they are just sketches, and thought experiments, where we "animate" the models in mind and in conversation. Just sketches, so less is invested. Less ego. Less time.

We sketch-prototype alternatives to try them out in the cheapest medium that fits what we're trying to understand and improve. We seek to probe, to learn, to verify the efficacy of the design elements we're considering, under multiple simultaneous demands. We acknowledge we can misperceive and deceive ourselves, and hold our work to scrutiny, seeing it from different perspectives, from different vantage points but also with different demands in mind. We consider and reconsider our design for fit to context, and to purpose. We evolve the design. We factor and refactor; we reify and elaborate. We test and evolve. We make trade-offs and judgment calls. We bring in others with fresh perspective to help us find flaws. We simulate. We figure out what to probe further, what to build and instrument. We bring what we can to bear, to best enable our judgment, given the concerns we're dealing with.

We humans are amazing; we invented and built all the tech we extend our capabilities with! And fallible; many failures, often costly, got us here, and we're still learning of and from unanticipated side-effects and consequences. Software is a highly cognitive substance with which to build systems on which people and organizations depend. So. We design-test our way, with different media and mediums to support, enhance, stress and reveal flaws in our thinking. Yes in code. But not only in code.

In the Cheapest Medium that Fits the Moment

Along the way — early, and then more as fits the moment — we're "mob modeling or "model storming" "out loud" "in pairs" or groups. And all that agile stuff. Just in the cheapest medium for the moment, because we need to explore options quick and dirty. Hack them — with sketches/models, with mock-ups, with just-enough prototypes. Not just upfront, but whenever we have some exploring to do, that we can do more cheaply than running experiments by building out the ideas in code. We do that too. Of course. But! We have the option to use diagrams and models to see what we can't, or what is hard to see and reason about, with (just) code. Enough early. And enough along the way so that we course correct when we need to. So that we anticipate enough. So that we direct our attention to what is important to delve into and probe/test further.

Whether we are seeking to learn more about users and needs, or more about design options, it is just as well
to bear the following in mind (so we try other routes, if they will yield the learning we need, more cheaply):

"Building software is an expensive way to learn" — Alistair Cockburn

Architecture decisions entail tradeoffs. We try for "and" type solutions, that give us more of more. More of the outcomes we seek, across multiple outcomes. Still, there are tradeoffs — "[architecting] strives for fit, balance and compromise among the tensions of client needs and resources, technology, and multiple stakeholder interests" (Rechtin and Maier).

Compromises mean not everyone is getting what they want (for themselves, or the stakeholders they see themselves beholden to serving) to the extent they want. Seeing the options and good ideas to resolve (only) the forces from the perspective of (just) a part, may.... lead to questioning design decisions ("throwing darts", in the terms of the cartoon) made from the perspective of the system. It can be hard to see why anyone would give up a good thing at a local level to benefit another part of the system, or to avoid precluding a future strategic option or direction. Sometimes those questions lead to a better resolution. Sometimes they just mean compromise.

Further, complex systems are, well, complex. Many, many! parts, in motion. In dynamic, and changing, contexts (of use, operations, evolutionary development). So there's uncertainty. And ambiguity. And lots of room for imperfect understanding. And mistakes. Will be made. Wrong turns taken. Reversed. Recovered from. We're fallible. So. More darts. Which is humbling — more so, if we're not humble enough to stay fleet, to try to learn earlier and respond to what we're learning.

All of which means we need to notice what is hard to notice from inside the tunnel of our own vision — where what we're paying attention to, shapes what we perceive and pay attention to.
I frequently quote the first of these heuristics ("expose your mental models to the open air"), but they are all important. We might summarize with "be humble." To me, confidence means being willing to act on my judgment, and humility means being willing to find myself wrong. Actionably willing, as in what Donella says — holding our models loosely; they're just models, and invite (kind, but discerning) scrutiny and alternatives.

“A change of perspective is worth 80 IQ points”
— Alan Kay

“A change of perspective is worth 80 IQ points” reminds us to take a different vantage point, to see from a different perspective. Consider the system from different points of view; use the lens of various views. This can play out multiple ways, but includes considering the design (structure, dynamics and implications) from the perspective of security, and from the perspective of responsiveness to load spikes, etc.

Another way to ensure a change of perspective, is to get another person's perspective. Invite peers working on other systems, say, to share what's been learned and seek out opportunities and weaknesses, things we missed. Our team can miss the gorilla, so to speak, when our attention is focused on the design issues of the moment. Fresh perspective, and even just naive questions about what the design means, can nudge an assumption or weakness into view. And merely telling the story, unfolding the narrative arc of the architecture...
to fit this person or audience, then that, gets us to adopt more their point of reference, across more perspectives — in anticipation, and when we listen, really listen, to their response and questions.

Relatedly, Minsky is further pushing us to the discipline of not just accepting our initial understanding, but rather seeking different understandings. These are the significant design decisions, decisions about the important stuff, after all.

"If you 'understand' something in only one way then you scarcely understand it at all. For then, if anything should go wrong, you'll have no other place to go. But if you represent something in multiple ways, then when one of them fails you can switch to another — until you find one that works for you."
— Marvin Minsky

We can understand the system as code, and building up mental models of the structure and "how it works." And we can understand the system (as we envision it, and as we build-evolve it) through sketches or visual models with reasoned arguments, explaining, exploring and defending the design. And we can understand the system through visualization of the system behavior, or through simulation or instrumentation (of prototypes, earlier, and the system in production, soon and then later, as the system is evolved). We can understand the system as perceived by someone new to the system, and as someone comfortable with the mental models we’ve imbued it with. We can understand the system directly and through the lens of an analogy (or hybrid blend of analogies). Understand it as loss and as gain. As a system we break down into parts, and as a system, or whole.

To change what we see, change where we see from, and what we see through: use different models, and aids to reasoning and seeing our thinking. State charts. Decision tables. Design structure matrices. Impact tables.
Fred Brooks wrote "Plan to throw one away. You will, anyway." I'd say: that too, but plan to throw several away — on paper. It's quick and cheap. Use rich pictures, use case and component diagrams and play over behavior of interest — repeat. Use other views. Do this at system level early to clarify direction worth taking/starting out in. But continue to do this (with just enough sketches and modeling focused on the concerns at hand) as challenges present themselves; some of these arise in the use context; some are internal "life sustaining" mechanisms that the system needs for it to be/become the kind of system it is (meet its persistence and dynamic data delivery/consistency needs; meet its scaling demands for spikes and growth; etc.). At any rate; "plan to throw some away," needs to include sketchprototypes. We need to try out alternatives in the cheapest medium we can learn more in; sometimes that's code, but not if a sketch will do. We don't learn at the granularity we learn when we learn in the medium of code, but we at least start to try ideas out, and explore and bat at them, investigate how they could work, in sketch-driven-dialog.

Three possibilities? For everything? That smacks of BDUF FUD (fear, uncertainty and doubt)?? Can't we just YAGNI that? Well, remember, these are make or break decisions. Game shapers and game changers. Still. We have more wisdom to call into play, this time Bucky Fuller's, and what Dana Bredemeyer calls the "extraordinary moment principle":

"What, at this extraordinary moment, is the most important thing for me to be thinking about?" — Buckminster Fuller

Architectural judgment factors.

"If you haven't considered more than one alternative, you're not designing" — Stuart Halloway
When we, as a field, for the most part turned away from BDUF (big design upfront) toward Agile methods, we tended, unfortunately, to turn away from architecture visualization and modeling. We've argued here that sketching and modeling is indeed a way to be agile — when it allows us to learn with the cheapest means that will uncover key issues, alternatives, and give us a better handle on our design approach. Not to do this as Big Modeling Upfront, but to have and apply our modeling skills early, to understand and shape strategic opportunity, and to set design direction for the system. And to return to modeling whenever that is the cheapest way to explore a design idea, just enough, just in time, to get a better sense of design bets worth investing more in, by building the ideas out and deploying at least to some in our user base, to test more robustly.

To return to a point we made earlier: in Agile, we're advancing both faces of design as we iterate, deliver working increments of value, get feedback, respond, and adapt. We're not just gaining a clearer understanding of the system; the system is evolving. Users adapt to the system, shifting work to the system or doing new things the system makes possible for them, and new possibilities for the system come into view. And using Agile practices like Test Driven Development and refactoring, we advance the design of the code. We evolve the system, and we evolve the design — in code, where that is good enough, and with diagrams and dialog and writing where that helps us create better designs, and communicate and evolve them.
This talk might have been subtitled “bringing visual design back to the agile table” — or “just enough, just in time visual design.” Where just enough varies by context, and its a judgment call (many judgment calls!). And the judgment needs to be that of the designers (including architects) not the “thought leaders” outside of the context.... That’s the upside of “just enough” — it has to be context-aware. The downside, in our rush-rush world, is that we might err on the side of not enough...

To recap, then, we’ve established -- okay, we’ve argued — that software architecture is that part of design that entails decisions that are structurally and strategically significant. That is, the integrity of the system depends on them. And more, they shape possibility for the business. Enable and constrain in make or break, business defining ways. At the level of scope of impact of the system in question, of course. These are decisions we need to probe more carefully. Probe in the sense of explore and understand, and probe in the sense of instrument. Visual design helps us expose our design ideas -- so we can see what we mean, run thought experiments across, collaborate to improve. Still, we need to write code, to fully probe, instrument, live-test the design in contexts of use and in operation. And we need to instrument our code, to see how well it is doing, from the perspective of code qualities.
Appreciation

I would like to thank the following for really helpful feedback that improved this work directly, or improved the Architecture Decisions post, and so this indirectly:

- Jessica Kerr
- Tristan Slominski

What was so wonderful about both Jessica and Tristan’s feedback is that they substantially improved the work, but without being in the least erosive.

And for encouragingly positive responses to the slides+notes in various stages of completeness:

- Nivia Henry
- Carlo Kuip
- Oliver Baier
- Jan Peuker
- John Brøndum
- Dan Klyn
- Guillermo Garcia

For conversations that improved my thinking in this area:

- Dawn Ahukanna

For catching specific glitches that the spellcheck can’t (gaol!) and other errors of commission and omission (I left out OOSE when I mentioned Booch and OMT!):

- Stephan Eggermont
- Hillel Wayne

And appreciation to Sasha Reid for this observation:

“This is an incredibly dense, detailed presentation! Two things stick - one: a skill of an architect is communicating problems in context and finding the boundaries of those contexts.. and that while code is design, it is an inefficient medium of communicating that design.”

And thank you to everyone who has supported with the implicit positive of a retweet!!

Finally, to Dana Bredemeyer, thank you for many years of working through this material and these ideas together.

All so very appreciated!