Vlad Ender
Solnet New Zeland Ltd.

The very first thing I noted when reviewing the list of patterns, was "Wow, there's tons of them!".
The Pattern Almanac alone lists 79 patterns. If I correctly remember one of the main drivers for patterns, it's making the communication easier.
I say "proxy" and you quite well know what I am talking about. We might clarify a some details, but we're not generally going to argue about semantics much.
79 patterns (or more, if I include other sources) is, in my opinion, too much to successfully use. If you look at the GoF patterns, arguably the most widely used in design, there's only 25 of them and I would even say that some (Factory and Abstract Factory to name one pair) are just variations on the theme.
The majority of other patterns out there are either specializations or combinations of these basics, or less commonly needed.
We can argue that software architects should be "smarter" than "normal" developers (whatever that would mean) and as such should be able to cope with more patterns than designers/developers need. Maybe, but they are still human beings and as such can cope only with a limited number of things. Moreover, even if someone could persuade me that all the architects are superhuman (I'd like that to be the case, believe me), I argue the
architectural patterns should not be a communication device for architects only.
In my ideal world, I should be able to talk to a development team in "we're using an event bus for this" and not having them suspecting me that I'm inviting them to a road trip. In fact, I consider this aspect even more important, because while developers generally communicate just between themselves, the role of an architect requires communication with a number of other people, sometimes with minimal technical knowledge.
Therefore, I consider a simplification of the pattern system a necessity for wide adoption.
To achieve this, I propose following techniques:

- abstract patterns. A number of patterns has the same basic idea, with slight variations on the theme. We should extract the common/abstract pattern. Then, at our leisure, we can "specialize" it further. A classic example is layers/components, etc. The PA list of patterns contains at least these patterns that have a common idea of splitting the system into parts with well defined responsibilities and dependencies - layers, layered architecture, N-tiered architecture, two and three-tiered architecture.
Another thing this would allow is to deal with the "what architecture?" problem. What I mean by this is that sometimes people start distinguishing between enterprise architecture, and (single) application architecture and product line architecture etc. While these are quite different, and undoubtedly do have patterns applicable to only some of them, there's a number of patterns that can apply to all - but your level of abstraction
is different.
For example, my favourite, Event Model. This can be used (together with metadata pattern) in an application for build-time dependencies decoupling ala Swing event listeners. Or I can use it in an enterprise, to build an enterprise event bus where all the applications chat together. The idea is very much the same, just what exactly the event
means is slightly different.

- define the abstract patterns so that we can combine them easily Patterns should avoid "concerns duplication". For example, I would argue that a microkernel pattern is really an extension of combination of facade and components/layers patterns (as presented above). This isn’t that important

though, as sometimes it makes good sense to name a common combination of patterns (MVC being a good example).

- weed out patterns that are analysis or design patterns.
This is hard. Especially if I take into account the point I made above, about abstract patterns. By applying the same logic, some patterns can be both design and architecture (and analysis), depending on the level of abstraction.
I have to admit that I don't entirely like the definition of an architectural pattern as presented. Mostly because it also covers analysis patterns to some extent (I think I could describe some enterprise business applications as "a set of predefined subsystems",
with "responsiblities" and "rules and guidelines of organizing the relationships" - i.e. CRM, GL, billing etc..).
What I think, based on my recent experience with justifying patterns as architectural, is that the problem being solved must be clearly based on one of the architectural goals. As architectural goals I see achieving a set of (prioritized) non-functional system qualities. Functional, directly business related decisions I would class as "design". But that's just my take on what the architecture means and your mileage may vary.
So, for example N-tiered clearly attempts to satisfy the quality of scalability and therefore is an architectural pattern.
Basing the patterns on this has an added benefit of the architect being able to go through his prioritized list of qualities, and find patterns that aim to satisfy them and do the trade-offs. As the reader knows by now, I love the decoupling event pattern, but sometime they just aren't the right choice. The required qualities and their priorities should tell me when this is the case.

- lastly, there should be only a small number (say that 25) of base patterns and they should be declared as the “foundation”. This would force us to prioritize patterns, most likely according to what the real world requirements are. For example, blackboard pattern is very interesting, but I don't think it's nearly as important as a component separation pattern.
As a start, these are the patterns (categorized by qualities) I would like to see there:
  - components
  - communication bus
  - cluster
  - facade
  - broker
  - pipes and filters
  - metadata
  - domain model
  - record lock (optimistic and pessimistic being specializations of the pattern).
  - model view controller
    ...