In this article, excerpted from our book, Reactive Application Development, we explain what is meant by “the big ball of mud” and talk about what developers can do to avoid that state.

The absence of proper application design has been called a “Big Ball of Mud” (Brian Foote and Joseph Yoder http://www.laputan.org/mud/), meaning a domain design that is “haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.” This is due to the fact that monolithic domain designs seldom, if ever, work and quickly become unmanageable. These types of designs are constantly in need of quick repairs that are not well thought out as part of the architecture as a whole, and as these accumulate, the system starts its inevitable decline.

What creates a ball of mud?
The forces that contribute to this ball of mud are:

- **Time**: This is the expectation that there isn’t enough time to do the best job or some mad rush to get software out to market or before a given season of the year.
- **Level of experience**: Insufficient programmer aptitude, inexperience or lack of supervision contribute to a ball of mud.
- **Cost**: This is the perception that higher quality software will bear too high a cost, which is interesting because most times inferior software costs substantially more than doing it right in the first place. Sometimes there just isn’t enough money to fund anything more than a hastily rushed project for a startup only concerned with fiscal survival.
- **Visibility**: Software, especially back-end software cannot be directly seen or touched. A messy user interface would draw criticism and immediate correction, whereas the backend can be built in shadows just waiting to fail.
- **Complexity**: This is a killer. Sometimes software needs to be somewhat complex to solve a problem, but when the software houses multiple complexities (bad encapsulation design) or is overly complex, it becomes confusing and unmanageable. Complex code is hard to look at, and discouraging to maintain.
- **Change**: Software changes; requirements change. If software is built in a tightly

For source code, sample chapters, the Online Author Forum, and other resources, go to www.manning.com/devore/
coupled fashion, without expectation of change there will be mud.

**What does the mudball look like?**
The big ball of mud is characterized by the following problems:

**THROWAWAY CODE**
Throwaway code is usually never really thrown away, as time is never taken to refactor it the right way because of the same short-term thinking that caused it in the first place. This is also known as the *sunk cost fallacy*, where there is an overwhelming and irrational feeling in management that so much money has been spent on a bad project that it’s too expensive to throw away.

**PIECEMEAL GROWTH**
This is growth over time and evolution. New York City is an excellent example of piecemeal growth of a city over time. The city started out organized, and then spread inward and outward from what used to be New Amsterdam, the present day Canal Street area up through Harlem. The cities streets are disorganized from downtown to midtown; Broadway even goes diagonal at some point. The city expanded according to need and not according to some grand plan. This may be characterized by the urban sprawl of your codebase. Los Angeles, another example of uncontrolled urban sprawl, is pictured below.
On one hand, a grand master plan might seem like it would result in a more organized city and code, but the reality is that all things change and planning on a moving target is setting yourself up for failure. On the other hand, growth without planning will end up as a mess, so what to do? The solution is atomicity of design, which means designing closely related sections of your system compartmentalized from other parts of the system. Keep the system up to date, by relentlessly refactoring locally.

KEEP IT WORKING
The software is important; your customers, workers, and indeed your money depends on it. Necessary improvements are desired, but not done for fear of breaking the system. Everyone from the top down is either afraid of having the code touched or touching it themselves. Very often the resulting code is throwaway, returning us to the throwaway code issues we just explored.
SWEEPING IT UNDER THE RUG
If you can’t make the dirt go away, you can hide it from plain sight. Unrealistic deadlines, insufficient requirements, and the feeling that just a little more hidden, dirty code won’t hurt anything and nobody will notice anyway right? It’s usually thought that this type of code is cost prohibitive, but the cost of maintaining such a codebase should not be ignored.

**How can you avoid the ball of mud?**

Here are some steps you can take to make sure you avoid the big ball of mud:

**CONTROL FRAGMENTATION WITH CONTINUOUS INTEGRATION**

When large teams work on a domain (even a team of three may be considered large), there is a potential for fragmentation due to the divergence of ideas within the team. Since the domain is continuously discovered, different developers may come up with different and divergent ideas. Why not solve this problem by further breaking down the domain? This is not the way to solve the problem. The domain is broken down to a level that still maps to a real life domain problem and when attempts are made to break it down further and artificially, that domain loses cohesion. So what to do?

*Continuous integration* to the rescue. Developers working on any given domain should come together at least daily, merging their code and ideas and, most importantly, keeping the ubiquitous language document updated with their evolved ideas. When code is merged often, emerged divergence is quickly guarded against and removed. As you can see, continuous integration allows close collaboration within the team.

**AVOID ANEMIC DOMAIN MODELS**

An anemic domain model is one that has not been thoroughly thought through in terms of domain-driven design and is an anti-pattern. This type of domain model at first glance might appear to seem reasonable, and map somewhat into real life objects, but when examined closely, it has no real behavior. An example of an anemic domain object would resemble a data structure with getters and setters rather than some entity with behavior and complex characteristics. These models do not fully benefit from object-oriented design in that the data and behavior is not encapsulated together, it’s more of a half step, and really matches a procedural style more than object oriented.

**DESIGN SHEARING LAYERS**

A ball of mud can be mitigated or prevented entirely by designing *shearing layers*. Let’s look at a building as an analogy to software. There is an argument that there isn’t anything that is really a building, but many layers of components such as foundation, wiring, roof, rafters, walls, etc. These components have their own rates of change and their own lifespans. This implies that software components should be grouped according to their own rates of change and lifespan. This applies to discrete areas of domain as well as areas of abstractions because abstractions change much slower than most other logic and should therefore exist in and of
itself and be maintained separately. A big ball of mud is the attempt to build the building without regards to layering.

**RECONSTRUCTION**

The only real cure for a ball of mud is *reconstruction*. Your system has declined to the point that it has unfortunately become a big ball of mud. Sometimes it is best to just throw it all away and start over. This is always a hard pill to swallow and is required by:

- Obsolescence of the tools and technology
- Long absence of original maintainers
- Real requirements emerged over time during the building of the throwaway system
- Such drastic design change as to render the original architectural assumptions useless

Architect’s most useful tools

"An architect’s most useful tools are an eraser at the drafting board, and a wrecking ball at the site.

- Frank Lloyd Wright