* Complexity is caused by state changes - We don’t like complexity just like we hate errors and bugs. In my opinion, complexity and errors have close connection that the more complexity a system has, the more chances the errors happen. In section 4.1, the authors state that complexity is caused by state, I agree to it. Usually when a system is deployed, the state is error free. However, as the time passes, the errors start showing up. The only difference is the state changes inside the system: components, values or even structures. Naturally, the more states change, the more errors may have. The authors also mentioned some daily experiences that restarts solve the problem because the system is back to the state when it was deployed where there are no errors. I think this is the reason why many applications have monthly maintenance.
* Impact of state on informal reasoning - In section 4.1.2, the authors mention that the state changes affect the testing at an exponential rate. I highly agree with it, the more complexity a system has, the more state combinations it has, and this number increases exponentially. It will take a great amount of extra time to test if those state combinations have errors if the system complexity is large, compared to a low complexity system.
* Object-Orientation Programming - I agree with the summary in section 5.1.3 that “OOP suffers greatly from both state-derived and control-derived complexity”(1). In my Java experience, the management of variables such as public, statc, private, and the derivation of the classes such as abstract class. Once the project size becomes large as well as the complexity, testing and managing takes many resources and time. Since the charactics of the OOP can’t be changed, reducing the complexity is a huge problem for both designers and developers.
* Functional Programming - I learned a lot from this week’s lecture about functional programming, as well as the discussion in this paper. I agree with the summary in section 5.2.5 that functional programming avoids the state-derived complexity. By focusing the result of the calculation instead of the calculation process, functions can be both input and output. Therefore, the state of the functions and components are stable, in order to reduce the complexity of the whole system. I like this logic fighting against the complexity since it is effective and easy to understand, and I hope I can learn and practice more functional programming in the future.
* Separation and the relationship between the components - I like the idea of this discussion in section 7.3.2. As long as the components are separated while only having minimum and necessary relationships. By having logic and state split, the complexity can be separated, measured and managed into smaller masses. As a whole system, it is easier to control the complexity and it also fits the popular design principles such as modularization. Functional programming is a good example for this idea since each function can be treated as a module and vice versa. Therefore, I think the discussions from authors are informative and enlightening.

References:

*“Out of the Tar Pit”*. Ben Moseley & Peter Marks. Feb 6, 2006.