1. Legal disclaimers:
   1. Buggy and incomplete as many POCs are (Many TODO’s across the code).
   2. Not focused on XAML looks, out-of-the box controls, didn’t implement XAML API for lack of time. It would make more sense implement based on existing ACE styles.
   3. Was able to implement many features discussed at the PowerPoint presentation: **Client (Presentation) Model, Mocking/Cache API, Navigation Services, Rules Engine, and Code Generator.**
   4. Not one of these feature is supposed to be implemented as is, but they were developed to show potential and some of them may overlap with existing features, but I wanted easy-of-development be the driver of this initiative.
   5. **This is not a meeting to discuss design, but to demonstrate the multiple ways that development productivity can be accelerated considerably. Let resist the temptation.**
   6. This is all original code with the exception of the BindableObject which I extended from ACE’s ViewModelBase.
2. **Presentation/Client Model**
   1. Singleton, always In-memory, Bi-directional graph where every Entity points to every other Entity that it relates to. Based on two main types: IndexedObservableCollection and EntityModel
   2. **Client Model’s Data should never be duplicated, EVER**. When different formats are needed it should be implemented through a calculated property (for instance, Fullname). Think Normalization.
   3. Services’ response should always be brought into the Client Model (never to the ViewModel). ViewModel consolidate entities and carry state that are solely required by the View. If needed someplace else that it must be a calculated property in the ClientModel (anecdote about XAML Converters).
   4. IndexedObservableCollection – Best of both worlds: Fast access of the Dictionary + change tracking of the ObservableCollection. Core of the Fidelity’s feature initiative.
   5. EntityModel – unit type of the IndexedObservableCollection. Expects a Key property. It is the foundation of many other features like IValid, IPersist and BindableObject.
   6. BindableObject – implements INotifyPropertyChanged and INotifyDirtyData (and soon INotifyPropertyChanging).
   7. These interfaces enforce extensibility points and lead engineers to the right decisions. **API should be an enforcer of good practices** and engineers should be able to infer what to do (only minimum training should be required). That is the reason for multiple layers. ViewModels should not be able to reach out to View components.
   8. Cache and Randomizers types for speed –Service calls development must be deployed from presentation development for quicker tests and Independent Deployment. It may require deployment Toggles. Randomizers provide easy way to mock all types of data.
   9. **Launch the App**
   10. Demonstrate how quickly search works. No coding was done towards performance. Not one task is asynchronous. Time permitting I will demonstrate AddressRelations Searches.
   11. Demonstrate how everything is updated at the same time without any extra code.
   12. Demonstrate how INotifyDirtyData works without any extra code.
   13. Explain how ICanPersist, IPersist, INotifyDirtyData and ViewModelBase.EditingEntity (which could be a composition of Entities) orchestrates a lot of work
   14. Explain how INotifyDirtyData supports multiple levels of UNDO, and implements Undo and Undo clear/reset at the time of Cancel or Save, again without any additional code.
3. **Navigation Services.**
   * 1. Should support Navigation from ViewModel to ViewModel, including parts (Controls or just GroupBoxes).
     2. Supports NavigateInto and NavigatingAway methods with arguments, which is the right place for Lazy initialization/Status reset and memory pressure relief, respectivelly. (PRISM supports that but it was outvoted).
     3. Support navigation history and shortcut keys should allow navigating back and forward, just like with browsers, for free.
     4. Navigation Services and/or IOC containers should determine the number of type instances allowed.
4. **Rules Engine**
   * 1. Dependent only on the ClientModel.
     2. Business rules anatomy: Triggers (events) + Conditional, time sensitive, versioned Boolean expression (RuleBase type) + Actions (ActionBase type).
     3. Rule is just a Boolean expression based on the ClientModel. Due to graph based nature of the ClientModel properties in multiple instances are available for comparison in one single rule (for instances Dates comparison).
     4. Rules can have different purposes (navigation, Validation, Services, pre-sales…)
     5. Actions are invoked based on the Rules result: True, False or Unknown (null).
     6. RuleActionBinding is the association of Rules with Actions.
     7. The rules invocation should not be done manually, but by a mechanism (Roslyn or PostSharp). Pause, to discuss PostSharp x Roslyn.
     8. The Rules Engine or a simple Rule can be paused, or re-started for while the app is still running. Exceptional control to the Stake Holders or App (not workstation) operators, and out-of-band development.
     9. Rules can be independently tested and make use of Mocking API.
     10. **Out-of-band Service calls and Rules development are precious features of this initiative. With that and code generation** the whole visual aspect of the app will be fully developed in few months.
5. **Code Generation**
   1. Many decisions about ACE’s visual presentation seems arbitrary. I don’t understand the reason for some many popups panels for instance. No need for them to be Modal (if they are), except for taking away chances of having out-of-sync data, which the ClientModel design change address.
   2. Visual design decisions must have a reason. We must identify patterns and code for the repetition. Exceptions may happen (Patterns that are not frequent enough to be worth of development).
   3. Demonstrate DataTemplates and explain the difference between using C# and Roslyn.
   4. Demonstrate how labels were calculated, in form and size. Explain how this should be part of another attribute, named Label (which already exist in .NET but must be extended)
6. **In Conclusion**
   1. The combination of all these features plus new XAML API and styles will boost the productivity considerably.
   2. But the law of diminishing returns are in full force here. We have to act with a sense of urgency.
   3. Also, as discussed, I recommend that we stop with Design-by-committee meetings. Design changes meetings with the whole team should be done to introduce the changes and explain re-factoring calculations.
   4. I’d like to be in-charge of these design changes and be responsible for their integration with Mike and Matt’s cooperation, in a position of Technical Architect.
   5. We have limited bandwidth for refactoring. If we handle multiple initiatives like the ones that I overheard for XAML we may integrate late. I recommend that you let me prioritize those because I may need the same professionals.