The main issue with a many-to-many relationship between observers and subjects is it can cause a lot of coupling issues. This can and increase the complexity of the architecture of a program and create a lot of dependencies that are hard to manage. We can try to prevent this by using the pull model. This will let our subjects notify the observers and then the needed observers will then run a get method to get the data as needed.

An advantage for the pull method is that it allows flexibility, and each observer can decide for itself what information to query without relying on the subject to send the correct information. However, this could also be seen as a disadvantage because the subject only sends a notification to the observers. So, the observers themselves must discover what has changed. This leads to frequent downcasting between multiple classes and quickly adds to code complexity.

To design this pattern, we use the standard observer diagram and add a dispatch class. This dispatch class is responsible for directly invoking the update method for all observers. The dispatch class will also be unique to each event and will handle most of the job the ConcreteSubject class used to handle. This makes it so the ConcreteSubject class to only knows how many observers are registered and the state of those observers. From here, we can add any number of subjects or observers and create a many-to-many relationship.