1)

- Para el proceso de resolución de un problema de desarrollo de software primero debe entender justamente el problema y su complejidad. Si esto no se cumple seguramente la solución no sea la adecuada o inclusive puede ser más compleja que el problema.

- Realizar diagramas de secuencia y/o colaboración para entender mejor el comportamiento del sistema.

- Luego a partir de todo lo anterior entramos en la etapa de “descubrimiento” que implica particionar el problema y hacer uso de abstracciones para dejar de lado las cosas que no interesan, hasta obtener el modelo del dominio.

- Luego al modelo de dominio le agregamos patrones de colaboración, reglas de negocio, test (esto se logra usando los diagramas mencionados anteriormente) y se obtiene el modelo de negocio.

- Por último, se validan las reglas de negocio, cuando las reglas de negocio estén validadas y los test no se rompan más, estamos en condiciones de realizar el diagrama de clases.

2)

Para el proceso de resolución de un problema de desarrollo de software usando programación orientada a prototipos necesitamos también hacer el análisis y descubrir los puntos importantes del problema. La diferencia es que el diseño no se deja para el final, no hace falta tener todo el modelo de negocio para hacer un diagrama de clases (no se usarían clases) sino que justamente se hace un prototipo inicial con cierta funcionalidad,se lo valida y se le va agregando funcionalidad a las entidades del sistema con el fin de pasar los test y cumplir con los requerimientos del usuario.

3) Con el uso del patrón **MVC** se habrá querido resolver un problema donde hay muchas vistas o interfaces de usuarios posibles y donde queremos tener separadas las 3 responsabilidades (representación de datos, gestión de eventos y lógica de negocio).
El patrón **Pipe-Filter** nos viene a solucionar el procesamiento de datos donde ese procesamiento se puede dividir en etapas, una detrás de la otra y al final de ellas se obtienen los datos originales pero transformados y/o filtrados.
El patrón **EA** implica un sistema complejo, con separación en tiers, probablemente una estructura cliente-servidor con muchos usuarios concurrentes y donde tenemos datos persistentes en las DB.

El patrón **LAYER** nos brinda distintas capas de abstracción donde cada una tiene una responsabilidad distinta, por ende imagino que es un sistema grande, donde se pueden agregar funcionalidad nueva y que esas funcionalidades (una en cada capa) se reutilizan para distintos casos.

4)

Viendo el diagrama de paquetes de la librería lo primero que cambiaría es el nombre “on/off”, no se entiende a que se refiere, le pondría “Interruptor on/off” ya que vamos a tener que agregar uno nuevo que sea “Interruptor por contacto”.
También debemos agregar “Agujereadora” por ende dentro de la librería me conviene tener 2 paquetes separados:
“Máquinas eléctricas” que tienen Agujereadora y Soplete.
“Interruptores” que tienen Interruptor on/off y Interruptor por contacto.
De esta manera podría usar el paquete de interruptores para otras cosas que no sean solamente las máquinas eléctricas.

Otra cosa que cambiaría es que Soplete no dependa directamente de Interruptor on/off ni Agujereadora de ambos interruptores porque si en un futuro se puede apagar de 200 maneras tendría 200 dependencias y para ver si está apagado tendría que revisar todas. Para esto agregaría un interfaz “Apagable” la cual implementan ambas máquinas eléctricas y una interfaz interruptor que la implementen todos los interruptores del paquete entonces la relación es entre interfaces.