**Diseño**

Su programa debe implementar el diseño del personal docente, que se describe a continuación. El diseño consta de 13 clases diferentes (6 como clases internas), 2 enumeraciones y 1 interfaz.

**edu.ncsu.csc216.product\_backlog.model.\***

edu.ncsu.csc216.product\_backlog.model.command. Subpaquete de edu.ncsu.csc216.product\_backlog.model que contiene la representación de objeto de un comando que un usuario realizaría en el sistema Product Backlog y que podría iniciar un cambio de estado. Este es un ejemplo del patrón de comando.

* Command. Encapsula la información sobre un comando de usuario que conduciría a una transición. Contiene una enumeración interna, que se proporciona en la sección Implementación, a continuación.
* Command Value. Una enumeración contenida en la clase Command. Representa uno de los seis comandos posibles que un usuario puede realizar para Task FSM.

edu.ncsu.csc216.product\_backlog.model.task. Subpaquete de edu.ncsu.csc216.product\_backlog.model que contiene la implementación del patrón de estado de la tarea FSM.

* Task. Clase concreta que representa una tarea rastreada en el sistema Product Backlog. Una tarea realiza un seguimiento de toda la información de la tarea, incluido el estado actual. La tarea es la clase de contexto del patrón State Design. El estado se actualiza cuando un Comando encapsula una entrada del usuario dada a la Tarea que conduce a una transición. Task encapsula la interfaz TaskState, seis clases \*State concretas y una enumeración:
* [TaskState interface](https://pages.github.ncsu.edu/engr-csc216/projects/project1/assets/code/TaskState.java) Interfaz que describe el comportamiento de clases TaskState concretas para la tarea FSM. No puede cambiar este código de ninguna manera. DEBE copiar este código de interfaz en la clase Task como una interfaz interna. Todas las clases \*State concretas deben implementar TaskState:
* BacklogState: clase concreta que representa el estado de trabajo pendiente de Task FSM.
* OwnedState: clase concreta que representa el estado de propiedad del Task FSM.
* ProcessingState: Clase concreta que representa el estado de procesamiento del Task FSM.
* VerifyingState: Clase concreta que representa el estado de verificación del Task FSM.
* DoneState: Clase concreta que representa el estado finalizado del Task FSM.
* RejectedState: Clase concreta que representa el estado cancelado del Task FSM.
* Type: una enumeración contenida en la clase Tarea. Representa las cuatro posibles categorías de tareas.

edu.ncsu.csc216.product\_backlog.model.product. Subpaquete de edu.ncsu.csc216.product\_backlog.model que contiene la clase Product.

* Product. Clase concreta que mantiene su nombre y una lista de Tareas asociadas.

edu.ncsu.csc216.product\_backlog.model.io. Subpaquete de edu.ncsu.csc216.product\_backlog.model que contiene clases de procesamiento de archivos.

* ProductReaders. Procesa un archivo que contiene información de productos y tareas y crea una Lista de Productos y sus Tareas asociadas.
* ProductsWriter. Escribe la lista dada de Productos en el nombre de archivo proporcionado.

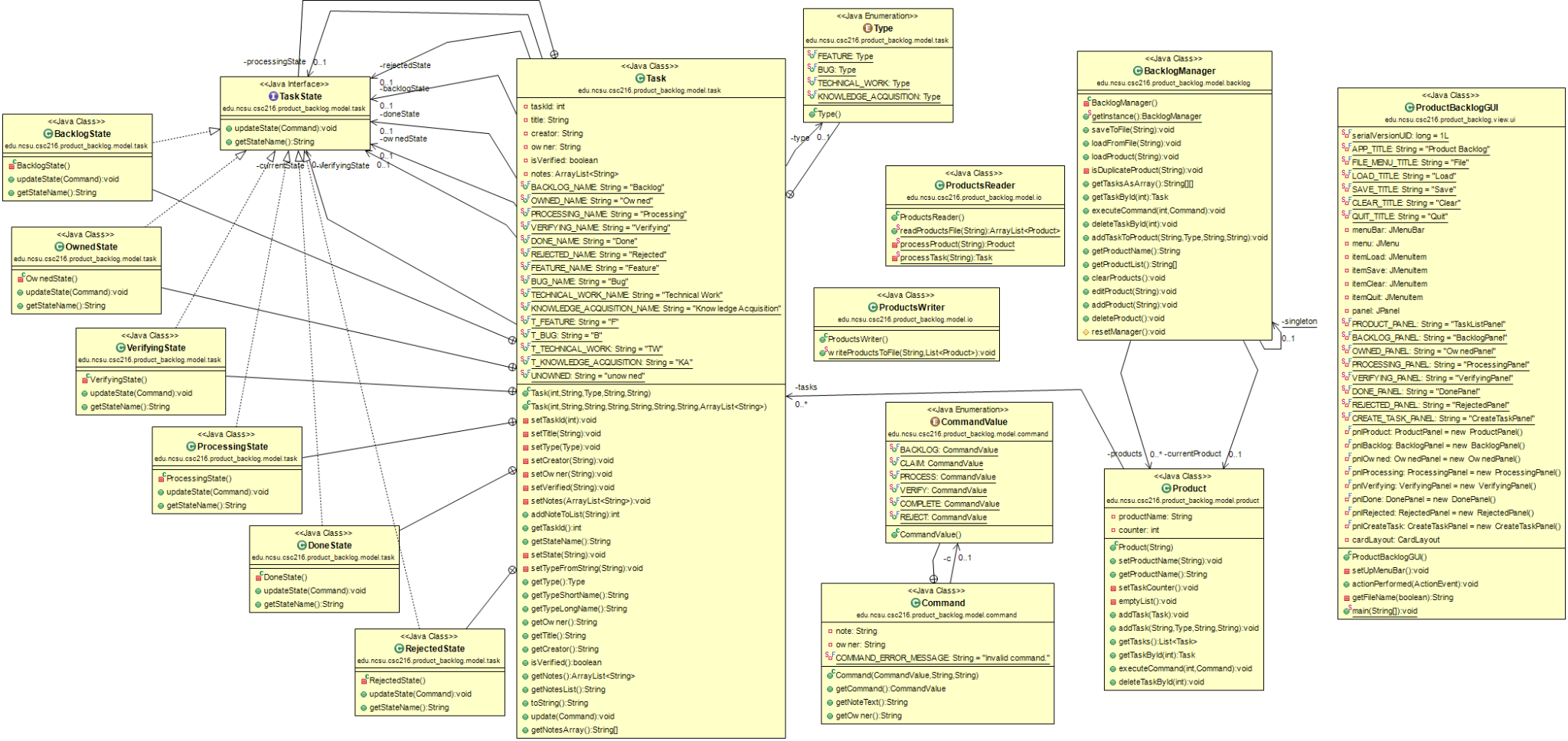
edu.ncsu.csc216.product\_backlog.model.backlog Subpaquete de edu.ncsu.csc216.product\_backlog.model que contiene

* BacklogManager.. Clase concreta que mantiene una lista de Productos, el Producto activo o actual, y maneja eventos desde la GUI. BacklogManager implementa el patrón de diseño Singleton.

**edu.ncsu.csc216.product\_backlog.view.\***

Edu.ncsu.csc216.product\_backlog.view.ui. Subpaquete de edu.ncsu.csc216.product\_backlog.view que contiene los elementos del controlador de vista del patrón Modelo-Vista-Controlador del Backlog del Producto.

* [ProductBacklogGUI class](https://pages.github.ncsu.edu/engr-csc216/projects/project1/assets/code/ProductBacklogGUI.java). La interfaz gráfica de usuario del proyecto. Esta es la clase que inicia la ejecución del programa. No puede cambiar este código de ninguna manera. ProductBacklogGUI interactúa con el resto del sistema a través del singleton BacklogManager.



**Notaciones de diagramas UML**

UML utiliza convenciones estándar para mostrar datos, métodos y relaciones. Muchos están ilustrados en el diagrama UML anterior. Aquí hay algunas cosas que debes tener en cuenta:

**- o un cuadrado rojo** frente a un miembro de la clase significa privado.

**+ o un círculo verde** frente a un miembro de la clase significa público.

**# o un triángulo amarillo** frente a un miembro de la clase significa protegido.

Los **miembros estáticos** (datos o métodos) están subrayados.

Los métodos que están declarados pero no definidos (métodos abstractos o declarados en interfaces) están en cursiva, a menos que estén en una interfaz.

Los **nombres de las clases abstractas** están en cursiva.

Las **flechas punteadas con cabezas triangulares** apuntan a las interfaces de las clases que las implementan.

Las **flechas sólidas con cabezas triangulares** van desde clases concretas hasta sus padres.

Las **flechas sólidas con cabezas simples** indican relaciones tiene-a (composición). La clase contenedora está en la cola de la flecha y la clase contenida está en la cabecera. La flecha está decorada con el nombre y el acceso del miembro de la clase contenedora. La flecha también está decorada con la "multiplicidad" de la relación, donde 0...1 significa que hay 1 instancia del miembro en la clase contenedora. (Consulte currentProduct en la flecha de BacklogManager a Product). Una multiplicidad de 0..\* significa que hay muchos, lo que generalmente indica una colección, como una matriz o una clase de colección. (Consulte las tareas en la flecha del Producto a la Tarea). Si el campo de la asociación debe ser estático, el nombre del campo está subrayado. (Consulte el campo singleton en BacklogManager.

Un **círculo que contiene una "X" o una cruz** se encuentra en el borde de una clase que contiene una clase interna (anidada), con una línea que la conecta con la clase interna. (Consulte la clase BacklogState y su correspondiente clase externa Task. Consulte también las enumeraciones).

Nuestro diagrama UML tiene alguna notación gráfica adicional:

Un **cuadrado rojo** (vacío o sólido) delante de un nombre significa privado. Los cuadrados sólidos son métodos y los cuadrados vacíos son miembros de datos. (Consulte Task.taskId.)

Un **círculo verde** (vacío o sólido) delante de un nombre significa público. (Consulte Producto.addTask().)

Un **diamante amarillo** (vacío o sólido) delante de un nombre significa protegido.

**SF** delante de un nombre significa estático, final. (Consulte Tarea.BACKLOG\_NAME).

Los **métodos adornados con C son constructores**. (Consulte Task.Task().) Los métodos privados adornados con una C son constructores en clases internas privadas. (Consulte BacklogManager.BacklogManager().) Tenga en cuenta que, si bien todas las clases requieren un constructor, no todos los constructores requieren implementación. En algunos casos, el constructor predeterminado es suficiente (por ejemplo, ProductsReader y ProductsWriter.

**Implementación**

Le sugerimos que implemente el Proyecto 2 Parte 2 una pieza a la vez. Esta sección proporciona detalles sobre la implementación, mensajes de error esperados para las excepciones y otra información útil para ayudarle a completar el Proyecto 1, Parte 2. ¡Esta sección proporciona las clases en el orden de implementación sugerido!

**Paquete edu.ncsu.csc216.product\_backlog.model.command.**

La clase **Command** crea objetos que encapsulan acciones (o entradas) del usuario que provocan transiciones al estado de una Tarea para actualizar. Le recomendamos que comience con la clase Comando porque la clase Tarea y los Estados concretos dependen de una clase Comando que funcione correctamente.

**Command** debe incluir un for para los posibles comandos (que corresponden a un clic o entrada de un botón por parte de un usuario) que pueden causar transiciones en su FSM (por ejemplo, CommandValue). Nuestro libro de texto define la enumeración como "un tipo que tiene sólo una pequeña cantidad de valores constantes predefinidos". Dado que hay un número discreto de acciones que un usuario puede realizar (esencialmente todos los botones que no están etiquetados como cancelar), una enumeración es bastante apropiada para enumerar esos valores.

Copie el siguiente código en la clase **Command** exactamente como se indica. Puede incluir el código de enumeración con los campos de **Command**. Las enumeraciones son pseudoobjetos (como matrices). Los constructores se enumeran en el diagrama de clases, pero no son necesarios en la implementación. Deberías Javadoc estas enumeraciones utilizando comentarios similares a los de nivel de clase.

**public enum CommandValue { BACKLOG, CLAIM, PROCESS, VERIFY, COMPLETE, REJECT }**

Para acceder a un valor en la enumeración, utilice el nombre de la enumeración seguido del valor (por ejemplo, CommandValue.BACKLOG). Los tipos enumerados son esencialmente un nombre dado a un valor entero. Por lo tanto, puede utilizar operadores de comparación primitivos (== y! =) para comparar variables de tipo enumerado del mismo tipo enumerado. Los tipos enumerados también pueden ser el tipo de una variable. Por ejemplo, el campo c de Command es de tipo CommandValue. Consulte la Sección 5.13 del Zybook para obtener más detalles sobre los tipos enumerados.

Un constructor de comandos tiene 3 parámetros: CommandValue c, propietario de la cadena, texto de la nota de cadena. Todos los parámetros son necesarios para cualquier comando. Cualquiera de las siguientes condiciones da como resultado una IllegalArgumentException al construir un objeto Command:

* Un **Command** con un parámetro CommandValue nulo. Un comando debe tener un CommandValue.
* Un **Command** con un texto de nota que es nulo o una cadena vacía.
* Un **Command** con un propietario nulo o una cadena vacía si CommandValue es CLAIM. Si se reclama una tarea, debe haber un propietario.
* Un **Command** con un propietario no nulo para cualquier otro valor de comando distinto de CLAIM. El propietario sólo es necesario para un comando CLAIM.

Los métodos restantes son captadores estándar para los campos. No hay métodos de establecimiento en Command. Los campos sólo se configuran durante la construcción.

**Paquete edu.ncsu.csc216.product\_backlog.model.task**

El paquete de task contiene el contexto (Task), el estado abstracto (la interfaz TaskState) y los estados concretos (todas las clases \*State) del patrón de diseño de Estado para la tarea FSM como parte del sistema Product Backlog.

Las seis clases concretas \*State implementan TaskState. TaskState debería ser una interfaz interna de Task. Las clases \*State deben ser clases internas de Task. Cada TaskState concreto debe admitir dos comportamientos: 1) actualizar cuando se le proporciona un command y 2) conocer su nombre.

**Type Enumeration**

Task incluye una enumeración de los posibles tipos de Task (por ejemplo,bug, feature, technical work, and knowledge acquisition). Dado que existe un número discreto de tipos de Task , una enumeración es bastante apropiada para enumerar esos valores. Utilice el siguiente código para la enumeración. Deberías Javadoc estas enumeraciones utilizando comentarios similares a los de nivel de clase.

**public enum Type { FEATURE, BUG, TECHNICAL\_WORK, KNOWLEDGE\_ACQUISITION }**

**Task Constants**

Task tiene las siguientes constantes, que son todas públicas. Esto significa que puedes usarlos directamente en otras clases y pruebas. ¡DEBEN ser públicos porque las pruebas del personal docente pueden depender de ellos!

Constantes para nombres de estados:

* BACKLOG\_NAME: una cadena constante para el nombre del estado del backlog con el valor de "Backlog".
* OWNED\_NAME: una cadena constante para el nombre del estado de propiedad con el valor de "Propiedad".
* PROCESSING\_NAME: una cadena constante para el nombre del estado de procesamiento con el valor de "Procesamiento".
* VERIFYING\_NAME: una cadena constante para el nombre del estado de verificación con el valor de "Verificar".
* DONE\_NAME: una cadena constante para el nombre del estado terminado con el valor "Listo".
* REJECTED\_NAME: una cadena constante para el nombre del estado rechazado con el valor de "Rechazado".

Constantes para type de nombres largos:

* FEATURE\_NAME: una cadena constante para el tipo de característica con un valor de "Característica".
* BUG\_NAME: una cadena constante para el tipo de error con un valor de "Error".
* TECHNICAL\_WORK\_NAME: una cadena constante para el tipo de trabajo técnico con un valor de "Trabajo técnico".
* KNOWLEDGE\_ACQUISITION\_NAME: una cadena constante para el tipo de adquisición de conocimientos con un valor de "Adquisición de conocimientos".

Constantes para nombres cortos de type:

* T\_FEATURE: una cadena constante para el tipo de característica con un valor de "F".
* T\_BUG: una cadena constante para el tipo de error con un valor de "B".
* T\_TECHNICAL\_WORK: Una cadena constante para el tipo de trabajo técnico con un valor de “TW”.
* T\_KNOWLEDGE\_ACQUISITION: Una cadena constante para el tipo de adquisición de conocimiento con un valor de “KA”.

Marcadores de posición constantes:

* UNOWNED: una cadena constante para una Tarea que no es de propiedad con un valor de "sin propiedad". Esto se utiliza para simplificar la E/S de archivos.

**Task Fields**

Task representa una tarea rastreada por nuestro sistema. Una tarea conoce su ID de tarea, estado actual, tipo, título, creador, propietario, está verificado y todas sus notas. El estado actual se actualiza a partir de los comandos propagados a la tarea desde la interfaz de usuario. Otros campos pueden actualizarse cuando se emite un comando según la definición de la tarea FSM. Los detalles de campo adicionales son:

* taskId: ID de una tarea.
* currentState: Estado actual de la tarea de tipo TaskState.
* tipo: Tipo de tarea de tipo Tipo (es decir, un valor de enumeración).
* título: Título de la tarea proporcionado por el usuario en el momento de la creación.
* creador: el nombre del usuario que creó la tarea tal como se proporcionó en el momento de la creación.
* propietario: El nombre del usuario propietario y responsable de resolver la tarea.
* isVerified: Verdadero si la tarea se ha verificado en la transición VerifyingA.
* notas: una ArrayList de cadenas que contiene la descripción de la tarea y cualquier nota agregada a medida que la tarea avanza a través del FSM.

Task mantiene una instancia de cada clase \*State concreta. Una de estas instancias puede ser el estado actual de la tarea en cualquier momento dado. Todas las instancias de \*State deben ser definitivas; no se modifican una vez construidos.

* backlogState: instancia final de la clase interna BacklogState.
* ownedState: instancia final de la clase interna OwnedState.
* ProcessingState: instancia final de la clase interna ProcessingState.
* verifyingState: instancia final de la clase interna VerifyingState.
* doneState: instancia final de la clase interna DoneState.
* RejectedState: instancia final de la clase interna RejectedState.

**Task Constructors**

Task tiene dos constructores:

* **Task(int id, String title, Type type, String creator, String note):** construye una tarea a partir de la identificación, el título, el creador y la nota proporcionados. Si alguno de los parámetros es una cadena nula o vacía, se lanza una excepción IllegalArgumentException. Si la identificación es cero o menos, se lanza una IllegalArgumentException. Se inicia una nueva tarea en backlogState. El propietario está configurado como SIN PROPIEDAD. La nota se agrega a la lista de notas recién creada.
* **Task (int id, String state, String title, String type, String creator, String owner, String verified, ArrayList<String> notes)**: construye una tarea proporcionando todos los valores de campo. Los campos de la Tarea se configuran con los valores de los parámetros después de la verificación de errores. Este constructor se utilizará en la clase ProductsReader. Puede resultarle útil crear métodos de configuración privados para configurar los campos. Este constructor verifica las restricciones de los objetos de Tarea enumerados en el [Formato de datos]. Le sugerimos que verifique el estado actual en último lugar para poder usar la información de otros parámetros para determinar si el estado proporcionado es apropiado para los otros parámetros. Por ejemplo, una tarea en BacklogState debe tener un valor de propietario de "Sin propietario".

Para ambos constructores, si hay un problema con cualquiera de los parámetros, se debe generar una IllegalArgumentException con el mensaje "Información de tarea no válida".

**Task Behaviors**

Task tiene captadores y definidores para manejar el trabajo con la información de campo. Todos los configuradores deben ser privados y son opcionales, pero muy recomendables. Los ambientadores privados están destinados a utilizarse durante la construcción de objetos. Si hay un problema con cualquiera de los parámetros para establecer métodos, se debe generar una IllegalArgumentException con el mensaje "Información de tarea no válida". Los captadores devuelven información que sería apropiada para mostrar en la GUI. El método setState(String) recibe un valor de cadena para el estado. Necesitará usar String para determinar qué instancia de \*State asignar al campo de estado para el estado actual. También puede verificar errores si el estado y otros parámetros son válidos según el [Formato de datos]. El método setTypeFromString() es similar a setState(). Los métodos getType\*Name() devolverán las cadenas cortas o largas para el tipo dado.

addNoteToList(String note) garantiza que la nota no sea nula ni una cadena vacía. Si la nota tiene contenido, entonces el nombre del estado entre corchetes se antepone a la nota: [Nombre del estado] nota....

El método toString() devuelve la representación de cadena de la Tarea que se imprime durante las operaciones de guardar archivos según el [Formato de datos]. NOTA: el guión inicial de los mensajes NO se almacena en la lista de notas. Se elimina antes de pasar desde archivos IO y se agrega en el método getNotesList(). Para las líneas que comienzan con '\*' y '-', hay un espacio después de esos caracteres. Cada línea debe terminar con un carácter '\n'.

Deberías comenzar creando los comportamientos principales de Task: constructores, definidores, captadores, toString() y los métodos especiales para trabajar con el estado, el tipo y agregar una nota a la lista. Una vez que estén funcionando, puede pasar al patrón Estado.

**Implementando el patrón de estado**

Para comenzar a implementar el patrón de estado subyacente de Task, primero agregue la interfaz TaskState como una interfaz interna privada de Task. Deberá copiar y pegar el código en la clase Tarea.

Task tiene seis clases internas y cada una implementa TaskState. Cada una de las clases internas representa los seis estados en la Tarea FSM. Cada TaskState concreto maneja la actualización cuando se le da un comando. Si el comando no es apropiado para el estado actual, entonces el método update() del TaskState concreto genera una excepción UnsupportedOperationException con el mensaje "Transición no válida". De lo contrario, el estado se actualiza según corresponda para el comando y según se define en la tarea FSM. Los campos de tareas adicionales se pueden modificar según las expectativas de comando y transición. Un TaskState concreto también conoce su nombre.

Una vez creados los estados concretos, puede finalizar la tarea implementando el método Task.update(Command). El método update(Command) controla la máquina de estados finitos delegando al método updateState(Command) del estado actual. El método update() genera una excepción UnsupportedOperationException si currentState determina que la transición, tal como la encapsula el comando, no es apropiada para el FSM. (Nota: Los métodos \*State.updateState(Command) en realidad crearán y lanzarán la excepción UnsupportedOperationException. El método Task.update(Command) permitirá que esa excepción pase a su llamador. Pero dado que la excepción se puede eliminar de Task.update (Comando), y es un método público, DEBE documentar el método en consecuencia).

**Paquete edu.ncsu.csc216.product\_backlog.model.product**

El paquete de productcontiene la clase Product. Un Product tiene un nombre de product, mantiene una Lista de Tareas y mantiene un contador que se utiliza para establecer los identificadores de nuevas Tareas cuando se agregan al Producto.

Al crear un nuevo Producto, establezca el nombre del producto y cree una nueva Lista vacía (puede usar una ArrayList o una LinkedList del Java Collections Framework como tipo construido). Además, el contador debe establecerse en 1. Se genera una excepción IllegalArgumentException si el valor del producto es nulo o es una cadena vacía. El mensaje de excepción debe ser "Nombre de producto no válido".

El producto admite las siguientes operaciones públicas:

* **setProductName()** que busca cadenas nulas y vacías y arroja una IllegalArgumentException con el mensaje "Nombre de producto no válido". si hay un problema.
* **getProductName()** devuelve el nombre del producto.
* **addTask(Task task)** agrega la tarea a la lista ordenada por ID (deberá escribir su propio algoritmo para manejar la clasificación; NO puede usar SortedList). La lista de tareas se mantendrá ordenada, por lo que podrá agregar una nueva tarea en orden al encontrar el índice apropiado para agregar la tarea. Si ya existe una tarea con la identificación proporcionada, se generará una excepción IllegalArgumentException con el mensaje "No se puede agregar la tarea". No olvide actualizar el contador después de que la tarea se haya creado y agregado correctamente en orden por identificación.
* **addTask (String title, Type type, String creator, String note)** creará una nueva tarea a partir de la información proporcionada. La identificación será el valor actual en el contador. No olvide actualizar el contador después de que la tarea se haya creado y agregado correctamente en orden por identificación.
* **getTasks()** devuelve la lista de tareas.
* **getTaskById(int id)** devuelve la tarea en la lista con la identificación proporcionada. Si no hay ninguna tarea con esa identificación, el método devuelve nulo.
* **deleteTaskById(int id)** elimina la tarea con la identificación proporcionada de la lista.
* **executeCommand(int id, Command c)** encontrará la Tarea con la identificación dada y la actualizará pasando el Comando dado.

También sugerimos el siguiente método de ayuda privada.

* **setTaskCounter()** encuentra la identificación de tarea más grande en la lista de tareas y establece el campo del contador en el máximo + 1. Si la lista está vacía, el contador se establece en 1. Este método es útil para garantizar que la próxima nueva tarea agregada a la El producto tiene el número de identificación correcto.

Cuando se trabaja con métodos que reciben un parámetro de identificación, no es necesario realizar una verificación de errores ni generar una excepción si la identificación no existe en la lista. Para getTaskById, devuelve nulo. Para todos los demás métodos, no cambie el estado interno de la lista.

**Paquete edu.ncsu.csc216.product\_backlog.model.io**

El paquete io contiene las clases que manejan la entrada y salida de archivos. Dado que los métodos son estáticos, no hay relaciones de composición explícitas modeladas en el diagrama de clases. Sin embargo, BacklogManager (y las pruebas del personal docente) dependerán de que estos archivos funcionen para probar BacklogManager además de probar las clases ProductsReader y ProductsWriter.

**ProductsReader**

ProductsReader tiene un método público readProductsFile que recibe una cadena con el nombre del archivo para leer. Si el archivo no se puede cargar porque no existe, el método generará una IllegalArgumentException con el mensaje "No se puede cargar el archivo". Se ignoran todos los productos o tareas no válidos (es decir, no se pueden construir, falta información o hay demasiada información para el artículo).

El archivo de entrada puede contener varios productos. Cada producto debe tener una o más tareas. Para ayudar con el procesamiento de la información, recomendamos el siguiente desglose del trabajo:

* Cree un objeto String y lea todo el archivo en el objeto String, línea por línea. Deberá agregar \n caracteres al final de cada línea, ya que se usan como delimitadores al leer línea por línea.
* Para dividir la línea en productos, cada uno de los cuales llamaremos token de producto, use \\r?\\n?[#] como delimitador. Esto separará el contenido mediante los caracteres de final de línea \n y \r y el siguiente carácter # que representa un producto. La primera línea del token del producto es el nombre del producto.
* Después de eliminar el nombre del producto del token de producto, puede usar el delimitador \\r?\\n?[\*] para dividir la cadena restante en tokens de tarea. La primera línea de un token de tarea es la información sobre la tarea, excepto la lista de notas. La primera línea está delimitada por caracteres de coma.
* Un token de tarea se puede dividir aún más en mensajes de notas usando el delimitador \\r?\\n?[-].
* Después de desglosar todo, puede crear objetos (o lanzar excepciones si hay un problema) y desarrollar las Tareas, agregarlas a Productos y agregar Productos a la lista de Productos para devolverlos a la persona que llama. Cuando sea posible, delegue la verificación de errores a los métodos de Tarea y Producto que serían más apropiados. Los productos se agregan a la lista de Productos en el mismo orden que el archivo de entrada. La clasificación de tareas por identificadores se producirá a medida que agregue las tareas al producto utilizando el método Product.addTask(Task) (suponiendo que haya implementado la clasificación correctamente).

Al trabajar en esta implementación, recomendamos utilizar declaraciones impresas para ver qué sucede en el archivo. Primero, intente dividir el contenido del archivo en productos e imprímalos para asegurarse de que sean correctos. Luego divida los productos en tareas. Finalmente, comienza a crear tareas y productos a partir de los tokens más pequeños. El método String.trim() ayudará a eliminar cualquier espacio en blanco inicial y final. Los caracteres iniciales #, \* y - deben eliminarse antes de agregar cualquiera de las cadenas a los objetos. El método String.substring() será útil para eso si es necesario.

**ProductsWriter**

ProductsWriter tiene un método público writeProductsToFile que recibe una cadena con el nombre del archivo para escribir y una lista de productos para escribir en el archivo. ProductsWriter debe usar el método toString() de Task para crear la salida con el formato adecuado para una tarea. Si hay errores o excepciones, se genera una IllegalArgumentException con el mensaje "No se puede guardar el archivo".

**Paquete edu.ncsu.csc216.product\_backlog.model.backlog**

BacklogManager implementa el patrón de diseño Singleton. Esto significa que solo se puede crear una instancia de BacklogManager. El patrón Singleton garantiza que todas las partes de ProductBacklogGUI interactúen con la misma instancia de BacklogManager en todo momento. Tenga en cuenta que ProductBacklogGUI no tiene una referencia global (o un campo) para BacklogManager. Dado que BacklogManager es un Singleton, la GUI puede acceder a esa única instancia en cualquier momento con una llamada a BacklogManager.getInstance() ya que getInstance() es un método estático.

Con un singleton, tenemos una instancia estática de la clase, que es singleton en BacklogManager. Esto se crea mediante una llamada al método estático getInstance(), que devuelve una instancia de BacklogManager. getInstance() comprobará si el singleton es nulo. Si el singleton es nulo, entonces getInstance() llama al constructor privado BacklogManager() para crear la instancia única. La instancia singleton siempre se devuelve; ¡La verificación nula está ahí para garantizar que siempre haya una! Y al hacer que el constructor sea privado, restringimos que los clientes creen instancias de BacklogManager.

Si queremos resetManager(), podemos establecer el singleton en nulo. Tenga en cuenta que resetManager() está protegido para restringir dónde se puede llamar. Está destinado a probar la clase BacklogManager.

Hay varios métodos que funcionan con Productos:

* **addProduct(String productName)**: Crea un nuevo Producto con el nombre dado y lo agrega al final de la lista de productos. Luego, el producto se carga como producto actual llamando al método loadProduct(String productName). Si ya existe un producto con el mismo nombre o si el nombre es nulo o una cadena vacía, se lanza una IllegalArgumentException con el mensaje "Nombre de producto no válido". Se deben ignorar las mayúsculas y minúsculas en las comparaciones del mismo nombre.
* **editProduct(String updateName)**: actualiza el nombre del producto actual al valor dado. Como parte de esto, se verifica que updateName no sea un duplicado de un producto existente o sea nulo o una cadena vacía. El producto conserva su ubicación en la lista de productos y sigue siendo el Producto actual. Si currentProduct es nulo al intentar editar, lanza una IllegalArgumentException con el mensaje "No se seleccionó ningún producto".
* **deleteProduct():** Elimina el Producto actual. El producto actual se actualiza al producto en el índice 0 de la lista o es nulo si no quedan productos. Si el producto actual es nulo al intentar eliminarlo, lanza una excepción IllegalArgumentException con el mensaje "No se seleccionó ningún producto".
* **loadProduct(String productName):** busque el producto con el nombre indicado en la lista, conviértalo en el producto activo o actual. Si el Producto solicitado no está en la lista, lanza una IllegalArgumentException con el mensaje "Producto no disponible".
* **getProductName():** Devuelve el nombre del producto para el producto actual. Si el producto actual es nulo, se devuelve nulo.
* **getProductList():** devuelve una matriz de cadenas de nombres de productos en el orden en que aparecen en la lista de productos. La GUI lo utiliza para completar el menú desplegable del producto.
* **clearProducts():** Restablece los productos a una lista vacía. El producto actual se establece en nulo.

**BacklogManager** trabaja con los archivos que contienen los Productos guardados y sus Tareas. Por lo tanto, BacklogManager trabaja en estrecha colaboración con las clases ProductsReader y ProductsWriter en los siguientes métodos.

* **loadFromFile (String fileName):** utiliza ProductsReader para leer el nombre de archivo dado. La lista de Productos devuelta se agrega al final de la lista de productos. El primer producto de la lista devuelto por ProductsReader se convierte en el Producto actual. Si hay otros productos en el sistema, se dejan como están.
* **saveToFile(String fileName):** si el proyecto actual es nulo o si no hay tareas en el proyecto actual, se debe generar una excepción IllegalArgumentException con el mensaje de error "No se puede guardar el archivo". De lo contrario, escriba Productss en el archivo usando la clase ProductsWriter.

BacklogManager también proporciona información a la GUI a través de métodos como getTasksAsArray() y getTaskById(). El método getTasksAsArray() devuelve una matriz de cadenas 2D que se utiliza para completar TaskTableModel (y la clase interna de ProductBacklogGUI.ProductPanel) con información. La matriz de cadenas 2D almacena [filas] [columnas]. La matriz debe tener 1 fila por cada tarea que necesite devolver. Debería haber 4 columnas:

* **Índice 0.** Número de identificación de la tarea
* **Índice 1**. Nombre del estado de la tarea
* **Índice 2.** Tipo de tarea (usando la cadena larga)
* **Índice 3**. Título de la tarea

Los otros métodos en BacklogManager delegan en los métodos del Producto con nombres similares. Recuerde verificar si currentProduct es nulo. Si es así, no hagas nada.

**Paquete edu.ncsu.csc216.product\_backlog.view.ui**

Una vez que haya completado el código del modelo que contiene la lógica del Product Backlog, agregará la clase ProductBacklogGUI.java al paquete edu.ncsu.csc216.product\_backlog.view.ui.

**Unit testing**

Sus pruebas JUnit deben seguir la misma estructura de paquete que las clases que prueban. Debe crear pruebas JUnit para todas las clases concretas que cree (incluso las clases internas; las prueba a través de la clase Task). Como mínimo, debe ejercitar todos los métodos de su solución al menos una vez en sus pruebas JUnit. Comience probando todos los métodos que no sean captadores simples, definidores simples o constructores simples para todas las clases que debe escribir y verifique que esté cubriendo todos los métodos. Si no es así, escriba pruebas para ejercitar métodos no ejecutados. Puede probar la funcionalidad común de una clase abstracta a través de una instancia concreta de su hijo.

Al probar métodos void, necesitará llamar a otros métodos que devuelvan algo para verificar que la llamada realizada al método void tuvo los efectos deseados.

Para cada método y constructor que genera excepciones, pruebe para asegurarse de que el método haga lo que se supone que debe hacer y genere una excepción cuando debería.

**Calidad de la prueba**

Durante la calificación, se inspeccionará la calidad de sus exámenes. Las pruebas de baja calidad pueden dar lugar a una deducción global del proyecto o posiblemente a una violación de la integridad académica dependiendo de la situación (por ejemplo, escribir código basura para lograr la cobertura; no hagas esto, lo buscamos). Las pruebas de baja calidad tienen una o más de las siguientes características (esta no es una lista exhaustiva):

Muy pocas afirmaciones (o ninguna) en un método de prueba, especialmente para las llamadas al código bajo prueba; esto generalmente se hace para obtener una alta cobertura sin verificar nada y es una práctica terrible. ¡No hagas esto!

Métodos de prueba que no hacen nada significativo.

Las pruebas de excepción carecen de una llamada fail() en las partes try o catch del bloque de excepción. Uno de los caminos debería fallar.

Clases de prueba que carecen de pruebas para métodos complejos en la clase bajo prueba. No es necesario tener métodos de prueba para cada captador y definidor, pero sí debe tener métodos de prueba para la funcionalidad compleja.

**Cobertura**

Para desbloquear los casos de prueba del personal docente, se requiere una cobertura del 60 % de declaraciones/líneas y del 80 % de cobertura de métodos para todas las clases que no sean UI.

Para obtener crédito completo en la parte de cobertura de su calificación, debe ejercer al menos el 80 % de las declaraciones/líneas y el 95 % de sus métodos en sus clases que no sean UI, excepto ProductsWriter. ProductsWriter solo requiere al menos un 70% de cobertura del estado de cuenta; es una clase corta y el camino de excepción es difícil de cubrir. Jenkins se creará para examinar estos umbrales a nivel global; sin embargo, debe observar la cobertura a nivel de clase como se informa en el resultado de la consola para asegurarse de cumplir con los umbrales de calificación enumerados.

Probablemente cubrirá los captadores, definidores y constructores simples como parte de las pruebas de funcionalidades más complejas en su sistema.

Le recomendamos que intente lograr una cobertura de condición cercana al 100% (donde cada predicado condicional se ejecuta tanto en la ruta verdadera como en la falsa para todas las rutas válidas en un método) en la mayor parte del programa.

**Probando el patrón de estado**

Dado que las clases \*State concretas son clases internas privadas, no podrá probarlas directamente. En su lugar, deberá actualizar una tarea en particular con un comando que provocará un cambio de estado (o no) y luego verificar el estado de la tarea y otros campos para asegurarse de que los valores se hayan actualizado correctamente.

Recuerde que al probar un FSM, desea probar cada transición fuera de cada estado. Eso significa que primero debe llegar al estado (emitiendo Comandos) y luego probar la transición (emitiendo otro Comando). Luego debes verificar que el estado de la Tarea sea correcto. Recomendamos utilizar el método afirmarTodo() para verificar todo el estado de la tarea a la vez. Esto le proporcionará una plantilla que luego podrá copiar, pegar y actualizar para sus pruebas.

Un escenario de prueba de ejemplo para probar la transición ProcessingB sería el siguiente:

* Crear una nueva tarea
* Verifique que la tarea recién creada sea un trabajo pendiente
* Emita un comando RECLAMACIÓN y asigne propietario "propietario" con la nota "asignación de propietario".
* Verifique que la tarea sea de propiedad con la información correcta.
* Emita un comando PROCESO.
* Verifique que la tarea esté en procesamiento con la información correcta.
* Emita un comando VERIFICAR.
* Verifique que la Tarea esté en Verificación con la información correcta.

Debe crear pruebas similares para todas las demás transiciones y luego considerar diferentes patrones de transición.

**Probando el singleton**

BacklogManager es una clase singleton, lo que significa que solo hay una instancia, y una vez que se crea esa instancia, esa es la instancia que se utiliza. Entonces, si una prueba agrega Productos a una lista, esos Productos estarán allí para la siguiente prueba. Para hacer que cada prueba sea atómica para que pueda ejecutarse de forma aislada (porque el orden de ejecución NO está garantizado), puede usar BacklogManager.resetManager() que debería establecer el singleton en nulo. Esto conducirá a la construcción de un nuevo singleton en la siguiente llamada a getInstance(). El constructor BacklogManager() creará una lista de Productos vacía y un Producto actual nulo. El método BacklogManager.resetManager() debe implementarse correctamente para que se ejecuten las pruebas del personal docente.