Guión para Video Explicativo: Sistema de Validación de Vigas en C#

Introducción

[Pantalla de Título: "Sistema de Validación de Vigas - Programación Orientada a Objetos"]
Narrador: ¡Hola! Bienvenidos a este tutorial donde aprenderemos sobre Programación Orientada a Objetos a través de un proyecto práctico: un sistema que valida si una viga puede soportar su propio peso.

[Mostrar diagrama simple de una viga con símbolos: %===*==]
Narrador: Imaginen que tenemos una viga construida con diferentes símbolos. Cada símbolo tiene un significado especial: Los símbolos **%**, **&** y **#** representan las bases que soportan la viga El símbolo **=** representa los largueros que forman la estructura El símbolo ***** representa las conexiones que unen las partes Nuestro programa debe determinar si la base puede soportar el peso total de la viga.

Desarrollo Parte 1: Conceptos Fundamentales

[Mostrar código de la clase abstracta BeamPart]
Narrador: Comencemos con el centro de nuestro programa: la **clase abstracta BeamPart**.

Una clase abstracta es como un molde o plantilla que define características comunes que compartirán todas las partes de nuestra viga.

Variables importantes:
- Symbol: guarda el carácter que representa la parte (%, &, #, =, *)
- Name: almacena el nombre descriptivo de la parte

Métodos abstractos:
- CalculateWeight: cada parte calculará su peso de manera diferente
- IsValidConnection: verifica si una parte puede conectarse con la anterior

La palabra **abstract** significa que esta clase no puede crear objetos directamente, sino que sirve como base para otras clases.

Desarrollo Parte 2: Las Clases Derivadas

[Mostrar código de la clase Base]
Narrador: Ahora veamos cómo las clases heredan de nuestra clase abstracta. Empezamos con la clase Base:

La clase Base **hereda** de BeamPart, lo que significa que automáticamente tiene las propiedades

Symbol y Name.

Nueva variable:
- Resistance: indica cuántas unidades de peso puede soportar esta base

El constructor recibe el símbolo de la base y asigna la resistencia correspondiente:
- % resiste 10 unidades
- & resiste 30 unidades
- # resiste 90 unidades

[Mostrar código de la clase Larguero]
Narrador: La clase Larguero representa las partes estructurales de la viga:
- override: significa que estamos dando nuestra propia implementación a los métodos abstractos
- CalculateWeight: el peso del larguero depende de su posición en la secuencia
- IsValidConnection: un larguero puede conectarse después de una base, otro larguero o una conexión

[Mostrar código de la clase Conexion]
Narrador: Finalmente, la clase Conexion:
Las conexiones tienen una regla especial: **solo pueden venir después de un larguero**, nunca después de otra conexión.

Desarrollo Parte 3: El Validador y los Cálculos

[Mostrar código de BeamValidator]
Narrador: La clase **BeamValidator** es nuestro "cerebro" que contiene toda la lógica de validación:

CreateBeamPart es una fábrica de objetos. Recibe un carácter y crea el objeto correspondiente. Es como un traductor que convierte símbolos en objetos de nuestro programa.

[Mostrar método CalculateTotalWeight]
El método más complejo es **CalculateTotalWeight**:
- totalWeight: acumula el peso total de toda la viga
- currentSequenceWeight: peso de la secuencia actual de largueros
- largueroPosition: posición del larguero en su secuencia (1, 2, 3...)

Lógica del cálculo:
1. Los largueros en secuencia pesan 1, 2, 3, etc.
2. Las conexiones (*) pesan el doble de la secuencia anterior
3. Cada conexión reinicia el contador de posición

Desarrollo Parte 4: El Programa Principal

[Mostrar código del Main]
Narrador: El programa principal coordina todo el sistema:

El método Main es el punto de entrada. Primero ejecuta casos de prueba automáticos, luego permite al usuario ingresar sus propias vigas.

[Mostrar método ProcessBeam]
ProcessBeam sigue un proceso ordenado:
1. Valida que la estructura sea correcta
2. Calcula el peso total
3. Compara con la resistencia de la base
4. Informa el resultado

Demostración Práctica

[Mostrar ejecución del programa]
Caso 1: "%" → resiste 10 unidades → soporta
Caso 2: "%=*" → resiste 10 unidades → soporta
Caso 3: "#===*==*=" → resiste 90 unidades → soporta
Caso 4: "%===*==*====" → resiste 10 unidades → NO soporta

Conclusión

[Mostrar diagrama de clases]
En este proyecto hemos aplicado conceptos fundamentales de la Programación Orientada a Objetos:
1. Abstracción
2. Herencia
3. Polimorfismo
4. Encapsulación

Ventajas del diseño:
- Modular
- Extensible
- Mantenible
- Reutilizable

Este sistema demuestra cómo la Programación Orientada a Objetos nos ayuda a modelar problemas del mundo real de manera organizada y eficiente.

¡Gracias por acompañarnos en este recorrido por el código!