

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Clase 14: Clean Code (Cap. 7)

Francisco Gazitúa Requena (fco_gazitua_requena@uc.cl)

IIC2113 Diseño Detallado de Software

25 de Septiembre, 2025

Error Handling

by Michael Feathers



Error Handling

"*Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, but if it obscures logic, it's wrong.*"

– Michael Feathers.

Error Handling

"Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, but if it obscures logic, it's wrong."

– Michael Feathers.

¿Por qué usamos excepciones mientras programamos?

Error Handling

"Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, but if it obscures logic, it's wrong."

– Michael Feathers.

¿Por qué usamos excepciones mientras programamos?

IIC2233: Para atajar errores lanzados por librerías.

Error Handling

"Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, but if it obscures logic, it's wrong."

– Michael Feathers.

¿Por qué usamos excepciones mientras programamos?

IIC2233: Para atajar errores lanzados por librerías.

IIC2113: Para que nuestro código sea más fácil de entender.

Error Handling

Mini-Repaso: Las excepciones nos ayudan a detener la ejecución de nuestro programa.

Error Handling

Mini-Repaso: Las excepciones nos ayudan a detener la ejecución de nuestro programa.

```
public class GameController{
    public bool HandleRound(){
        try{
            tryToHandleTurn();
            Console.WriteLine("Turn handled successfully.");
        }
        catch (GameEndedException e){
            //Handle end of turn
        }
    }

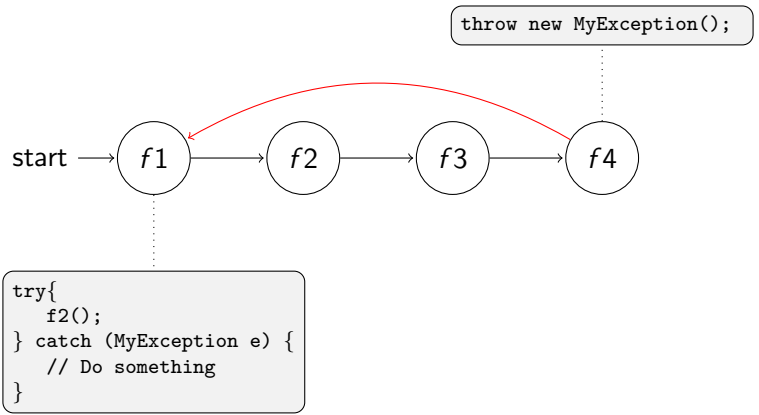
    public void tryToHandleTurn(){
        // Handle turn logic
        if (WasTeamWipedOut()){
            throw new GameEndedException("End of round reached.");
        }
    }
}
```

Error Handling

Mini-Repaso: Las excepciones pueden detener una cadena muy grande de ejecución.

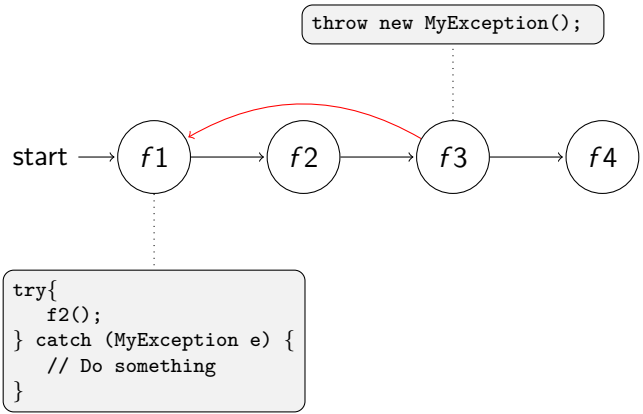
Error Handling

Mini-Repaso: Las excepciones pueden detener una cadena muy grande de ejecución.



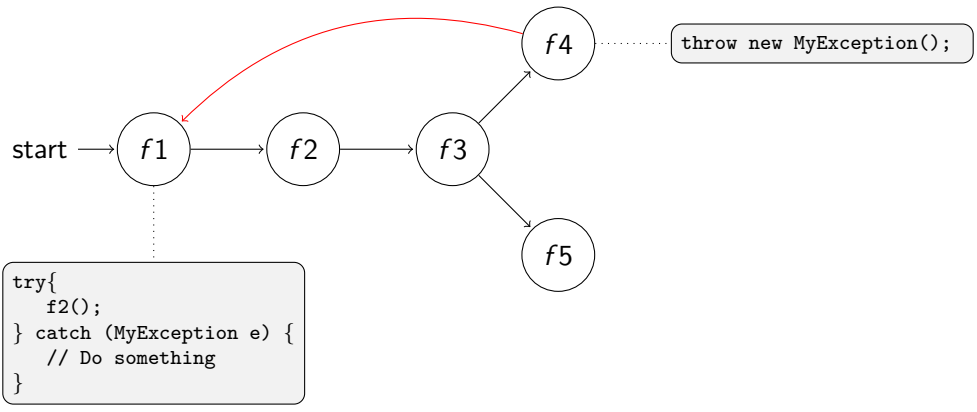
Error Handling

Mini-Repaso: Las excepciones pueden detener una cadena muy grande de ejecución.



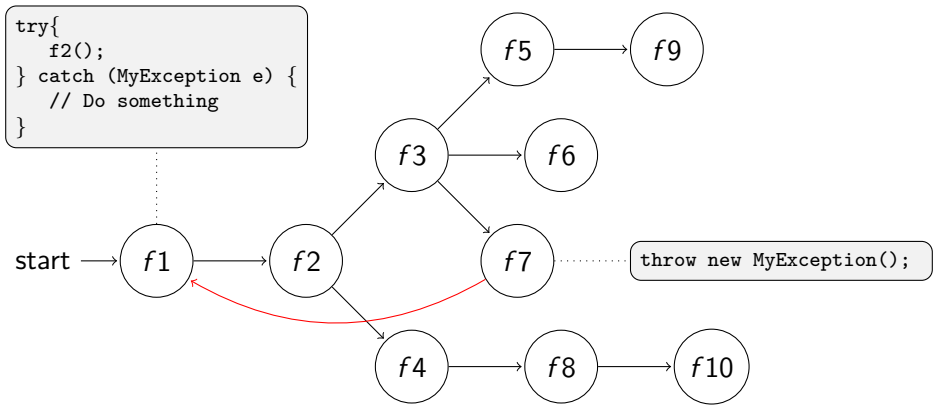
Error Handling

Mini-Repaso: Las excepciones pueden detener una cadena muy grande de ejecución.



Error Handling

Mini-Repaso: Las excepciones pueden detener una cadena muy grande de ejecución.



Error Handling

Mini-Repaso: Podemos crear nuestras propias excepciones

```
public class MyCustomException : ApplicationException
{
    public MyCustomException() : base() { }

    public MyCustomException(string message) : base(message) { }
}
```

```
// Lanzar una excepción de forma simple
throw new MyCustomException();

// Lanzar una excepción con un mensaje
throw new MyCustomException("Este es un mensaje de error personalizado
    .");
```

Error Handling

Mini-Repaso: Podemos crear nuestras propias excepciones

```
public abstract class MyAbstractCustomException : ApplicationException
{
    public abstract string GetErrorMessage(ProgramData data);
}

public class MyConcreteException : MyAbstractCustomException
{
    public override string GetErrorMessage(ProgramData data)
        => $"Error occurred in program: {data.ProgramName} with
    version: {data.Version}";
}
```

Error Handling

Mini-Repaso: Podemos crear nuestras propias excepciones

```
try
{
    // Código que puede lanzar una excepción
}
catch (MyAbstractCustomException ex)
{
    Console.WriteLine(ex.GetErrorMessage(_programData));
}
```

Error Handling

Recomendación 1: Usa excepciones, no códigos de error.

Error Handling

Recomendación 1: Usa excepciones, no códigos de error.

```
2  public void sendShutDown() {
3      DeviceHandle handle = getHandle(DEV1);
4      // Check the state of the device
5      if (handle != DeviceHandle.INVALID) {
6          retrieveDeviceRecord(handle);
7          // If not suspended, shut down
8          if (record.getStatus() != DEVICE_SUSPENDED) {
9              pauseDevice(handle);
10             clearDeviceWorkQueue(handle);
11             closeDevice(handle);
12         }
13         else
14             logger.log("Device suspended. Unable to shut down");
15     }
16     else
17         logger.log("Invalid handle for: " + DEV1.toString());
18 }
```

Error Handling

Recomendación 1: Usa excepciones, no códigos de error.

```
2  public void sendShutDown() {
3      try {
4          tryToShutDown();
5      } catch (DeviceShutDownError e) {
6          logger.log(e);
7      }
8  }
9
10 private void tryToShutDown() {
11     DeviceHandle handle = getHandle(DEV1);
12     DeviceRecord record = retrieveDeviceRecord(handle);
13     pauseDevice(handle);
14     clearDeviceWorkQueue(handle);
15     closeDevice(handle);
16 }
```

Error Handling

Recomendación 2: A veces no es necesario agregar una excepción.

```
2 try {
3   MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
4   m_total += expenses.getTotal();
5 } catch(MealExpensesNotFound e) {
6   m_total += getMealPerDiem();
7 }
```

Error Handling

Recomendación 2: A veces no es necesario agregar una excepción.

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

Solución: Usar el patrón `Special Case`

```
public class PerDiemMealExpenses : MealExpenses {
  public override int getTotal() {
    // return the per diem default
  }
}
```

Error Handling

Recomendación 2: A veces no es necesario agregar una excepción.

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

Solución: Usar el patrón `Special Case`

```
public class PerDiemMealExpenses : MealExpenses {
  public override int getTotal() {
    // return the per diem default
  }
}
```

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Error Handling

Ejemplo: Digamos que modelas las habilidades de un juego usando herencia. Toda habilidad hereda de `Skill` y tiene un método `Apply(...)` para ejecutar el efecto de la habilidad. Luego necesitas una forma de acceder a las habilidades según su nombre. Digamos que usamos una clase llamada `SkillCatalog` para ello.

Error Handling

Ejemplo: Digamos que modelas las habilidades de un juego usando herencia. Toda habilidad hereda de `Skill` y tiene un método `Apply(...)` para ejecutar el efecto de la habilidad. Luego necesitas una forma de acceder a las habilidades según su nombre. Digamos que usamos una clase llamada `SkillCatalog` para ello.

```
1  Class SkillCatalog
2  {
3      private Dictionary<string,Skill> _skills;
4
5      public Skill GetSkillBy(string name)
6      {
7          if(_skills.ContainsKey(name))
8              return _skills[name];
9          throw new NotImplementedException();
10     }
11     /* ... */
12 }
```

Error Handling

Luego tenemos un método que para usar una habilidad la obtiene desde el catálogo:

```
1  private void ApplySkill(string skillName)
2  {
3      try
4      {
5          Skill skill = _skillCatalog.GetSkillBy(skillName);
6          skill.Apply(/* ... */);
7      }
8      catch(NotImplementedException e)
9      {
10         // we ignore this case
11     }
12 }
```

Error Handling

Luego tenemos un método que para usar una habilidad la obtiene desde el catálogo:

```
1  private void ApplySkill(string skillName)
2  {
3      try
4      {
5          Skill skill = _skillCatalog.GetSkillBy(skillName);
6          skill.Apply(/* ... */);
7      }
8      catch(NotImplementedException e)
9      {
10         // we ignore this case
11     }
12 }
```

... pero en este caso nos podemos ahorrar la excepción.

Error Handling

Creemos una clase con un habilidad vacía.

```
Class EmptySkill : Skill
{
    public override void Apply(/* ... */)
    {
        // Nothing happens when we apply this skill
    }
}
```

Ahora podemos retornar la habilidad vacía en vez de tirar una excepción.

Error Handling

```
Class SkillCatalog
{
    private Dictionary<string,Skill> _skills;

    public Skill GetSkillBy(string name)
    {
        if(_skills.ContainsKey(name))
            return _skills[name];
        return new EmptySkill();
    }
    /* ... */
}
```

Error Handling

Como resultado, el código para ejecutar una habilidad queda más limpio:

```
1 private void ApplySkill(string skillName)
2 {
3     Skill skill = _skillCatalog.GetSkillBy(skillName);
4     skill.Apply(/* ... */);
5 }
```

Error Handling

Recomendación 3: NO retornes `null`.

Error Handling

Recomendación 3: NO retornes `null`.

```
2  public void registerItem(Item item) {
3      if (item != null) {
4          ItemRegistry registry = peristentStore.getItemRegistry();
5          if (registry != null) {
6              Item existing = registry.getItem(item.getID());
7              if (existing.getBillingPeriod().hasRetailOwner()) {
8                  existing.register(item);
9              }
10         }
11     }
12 }
```

... básicamente retornar `null` es igual de malo que usar códigos de error.

Error Handling

En el `SkillCatalog` era posible retornar `null` en vez de tirar una excepción:

Error Handling

En el `SkillCatalog` era posible retornar `null` en vez de tirar una excepción:

```
Class SkillCatalog
{
    private Dictionary<string,Skill> _skills;

    public Skill GetSkillBy(string name)
    {
        if(_skills.ContainsKey(name))
            return _skills[name];
        throw new NotImplementedException();
    }
    /* ... */
}
```

Error Handling

En el `SkillCatalog` era posible retornar `null` en vez de tirar una excepción:

```
Class SkillCatalog
{
    private Dictionary<string,Skill> _skills;

    public Skill? GetSkillBy(string name)
    {
        if(_skills.ContainsKey(name))
            return _skills[name];
        return null;
    }
    /* ... */
}
```

Error Handling

En el `SkillCatalog` era posible retornar `null` en vez de tirar una excepción:

```
1  private void ApplySkill(string skillName)
2  {
3      try
4      {
5          Skill skill = _skillCatalog.GetSkillBy(skillName);
6          skill.Apply(/* ... */);
7      }
8      catch(NotImplementedException e)
9      {
10         // we ignore this case
11     }
12 }
```

Error Handling

En el `SkillCatalog` era posible retornar `null` en vez de tirar una excepción:

```
1 private void ApplySkill(string skillName)
2 {
3     Skill? skill = _skillCatalog.GetSkillBy(skillName);
4     if(skill != null)
5         skill.Apply(/* ... */);
6 }
```

Error Handling

En el `SkillCatalog` era posible retornar `null` en vez de tirar una excepción:

```
1 private void ApplySkill(string skillName)
2 {
3     Skill? skill = _skillCatalog.GetSkillBy(skillName);
4     if(skill != null)
5         skill.Apply(/* ... */);
6 }
```

... pero sigue siendo más limpio tener la clase `EmptySkill`.

```
1 private void ApplySkill(string skillName)
2 {
3     Skill skill = _skillCatalog.GetSkillBy(skillName);
4     skill.Apply(/* ... */);
5 }
```

Error Handling

Recomendación 4: NUNCA pases `null` como argumento.

Error Handling

Recomendación 4: NUNCA pases `null` como argumento.

```
8  public class MetricsCalculator {
9      public double xProjection(Point p1, Point p2) {
10         if (p1 == null || p2 == null) {
11             throw InvalidArgumentException(
12                 "Invalid argument for MetricsCalculator.xProjection");
13         }
14         return (p2.x - p1.x) * 1.5;
15     }
16 }
```

... la vida se vuelve una tortura si permitimos que los argumentos sean `null`.

Resumen

En la Clase de Hoy

(No tan) Sorprendentemente, las excepciones nos pueden ayudar a limpiar nuestro código.

En la Clase de Hoy

(No tan) Sorprendentemente, las excepciones nos pueden ayudar a limpiar nuestro código.

Idea central: Tirar excepciones nos permite separar la lógica para manejar errores del flujo normal de nuestra aplicación.

En la Clase de Hoy

(No tan) Sorprendentemente, las excepciones nos pueden ayudar a limpiar nuestro código.

Idea central: Tirar excepciones nos permite separar la lógica para manejar errores del flujo normal de nuestra aplicación.

Esto deriva en 4 consejos claves:

1. Prefiere excepciones a retornar códigos de error.
2. De ser posible usa el patrón `Special Case` para evitar lanzar una excepción.
3. No retornes `null`.
4. No recibas `null` como parámetro.

En la Clase de Hoy

(No tan) Sorprendentemente, las excepciones nos pueden ayudar a limpiar nuestro código.

Idea central: Tirar excepciones nos permite separar la lógica para manejar errores del flujo normal de nuestra aplicación.

Esto deriva en 4 consejos claves:

1. Prefiere excepciones a retornar códigos de error.
2. De ser posible usa el patrón `Special Case` para evitar lanzar una excepción.
3. No retornes `null`.
4. No recibas `null` como parámetro.
5. Hay que utilizar Excepciones personalizadas.
6. Hay que minimizar el número de catchs.

¿Cuáles son los descuentos típicos en el proyecto debido al cap. 7?

Error Handling

Descuentos más comunes:

- Retornar códigos de error (e.j., `null`) en vez de usar excepciones. Notar que si se retorna un `bool` que actúa como código de error, igual se podría descontar.
- Se lanzan/atajan excepciones generales (e.j., `ApplicationException`) en vez de crear excepciones personalizadas.
- Funciones que retornan o reciben `null`.
- Cada bloque de `try-catch` debería tener un solo `catch`. Son muy pocas la veces que se justifica tener varios `catchs`.
- Usar excepciones (o códigos de error) cuando un *Special Case Pattern* bastaba.

Error Handling

Descuentos más comunes:

- Retornar códigos de error (e.j., `null`) en vez de usar excepciones. Notar que si se retorna un `bool` que actúa como código de error, igual se podría descontar.
- Se lanzan/atajan excepciones generales (e.j., `ApplicationException`) en vez de crear excepciones personalizadas.
- Funciones que retornan o reciben `null`.
- Cada bloque de `try-catch` debería tener un solo `catch`. Son muy pocas la veces que se justifica tener varios `catchs`.
- Usar excepciones (o códigos de error) cuando un *Special Case Pattern* bastaba.

... les recomiendo fuertemente que lean este capítulo :)

Actividad