Refactoring Away from Exceptions

Vladimir Khorikov
PROGRAMMER

@vkhorikov www.enterprisecraftsmanship.com

Exceptions and Readability

```
public ActionResult CreateEmployee(string name) {
    try {
        ValidateName(name);
        // Rest of the method

        return View("Success");
    }
    catch (ValidationException ex) {
        return View("Error", ex.Message);
    }
}

private void ValidateName(string name) {
    if (string.IsNullOrWhiteSpace(name))
        throw new ValidationException("Name cannot be empty");

    if (name.Length > 100)
        throw new ValidationException("Name is too long");
}
```

Exceptions and Readability

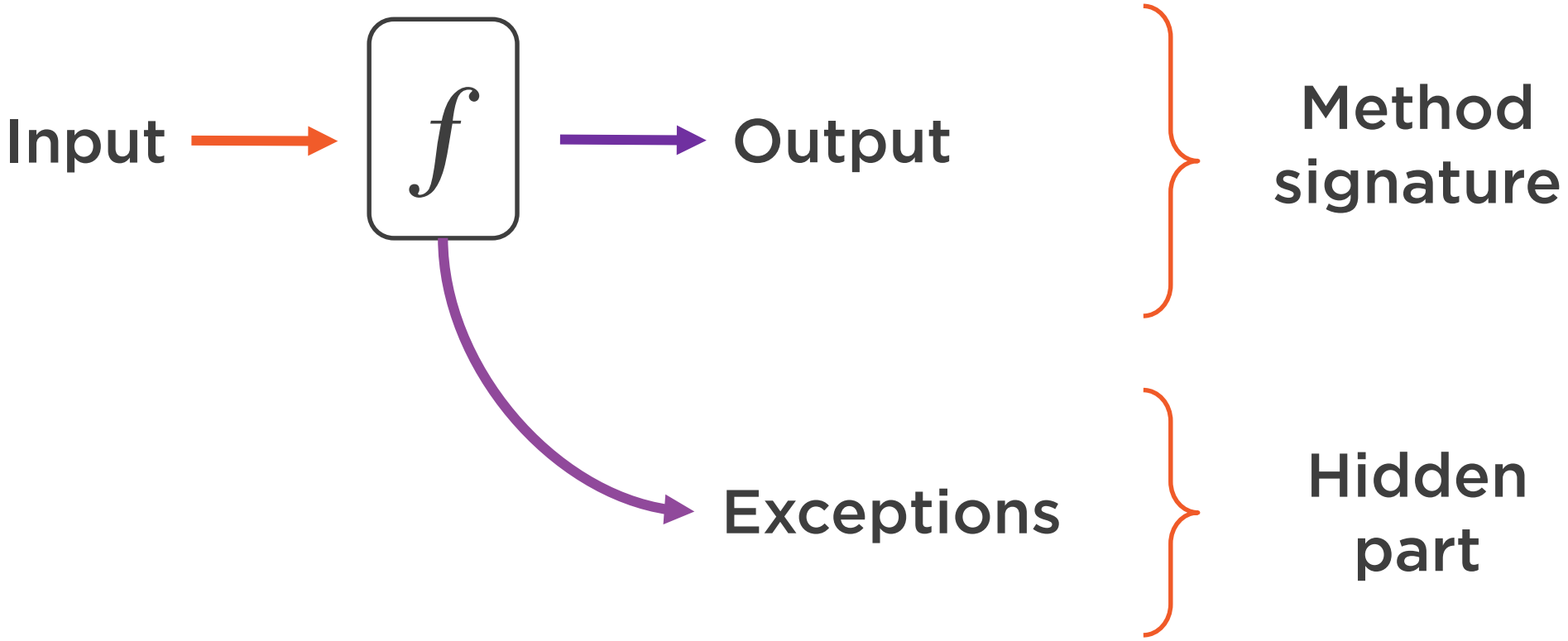
```
public Employee CreateEmployee(string name)
{
    ValidateName(name);

    // Rest of the method
}
```

Exceptions and Readability

Method with exceptions ≠ **Mathematical function**

Exceptions for flow control = **Goto statements**

Exceptions and Readability



Exceptions and Readability

```
public ActionResult CreateEmployee(string name)
{
    string error = ValidateName(name);
    if (error != string.Empty)
        return View("Error", error);

    // Rest of the method
    return View("Success");
}

private string ValidateName(string name)
{
    if (string.IsNullOrWhiteSpace(name))
        return "Name cannot be empty";

    if (name.Length > 100)
        return "Name length cannot exceed 100 characters";

    return string.Empty;
}
```

Always prefer using return values over exceptions.

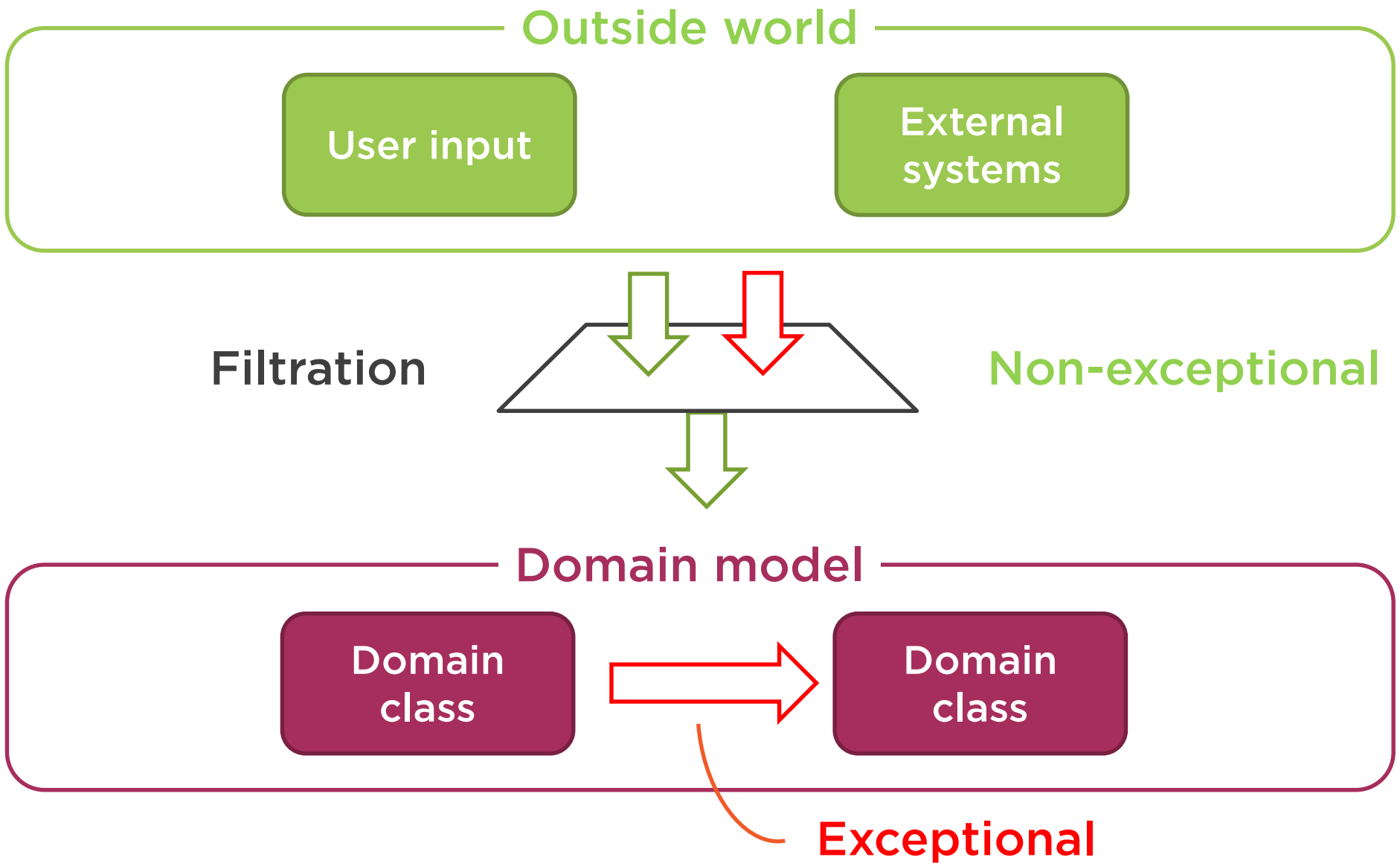
Use Cases for Exceptions

- Exceptions are for exceptional situations
- Exceptions should signalize a bug
- Don’t use exceptions in situations you expect to happen

Use Cases for Exceptions

Validations  **Exceptional situation**

Use Cases for Exceptions



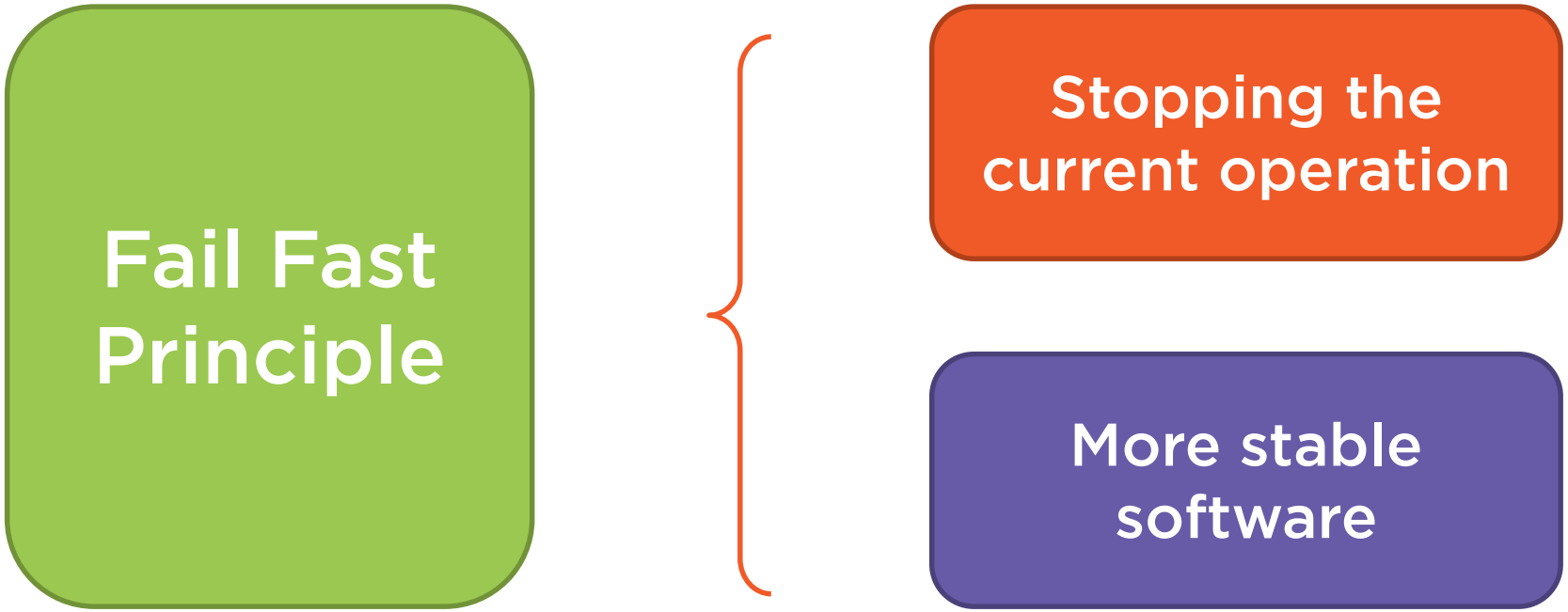
Use Cases for Exceptions

```
public ActionResult UpdateEmployee(int employeeId, string name)
{
    string error = ValidateName(name);
    if (error != string.Empty)
        return View("Error", error);

    Employee employee = GetEmployee(employeeId);
    employee.UpdateName(name);
}

public class Employee
{
    public void UpdateName(string name)
    {
        if (name == null)
            throw new ArgumentNullException();

        // Rest of the method
    }
}
```

Fail Fast Principle

How to handle exceptions?

Fail Fast Principle



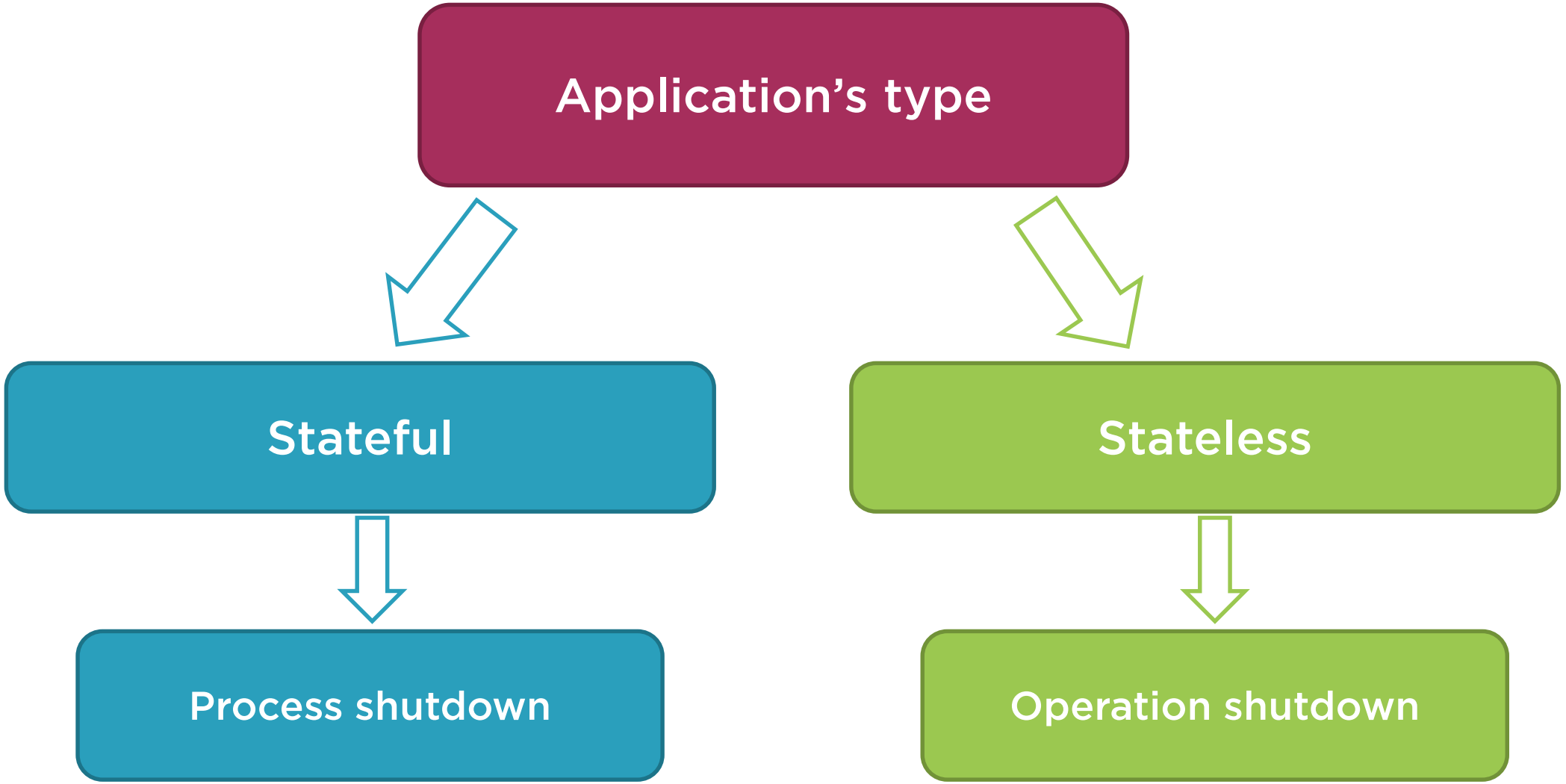
Fail Silently

```
public void ProcessItems(List<Item> items)
{
    foreach (Item item in items)
    {
        try
        {
            Process(item);
        }
        catch (Exception ex)
        {
            Logger.Log(ex);
        }
    }
}
```

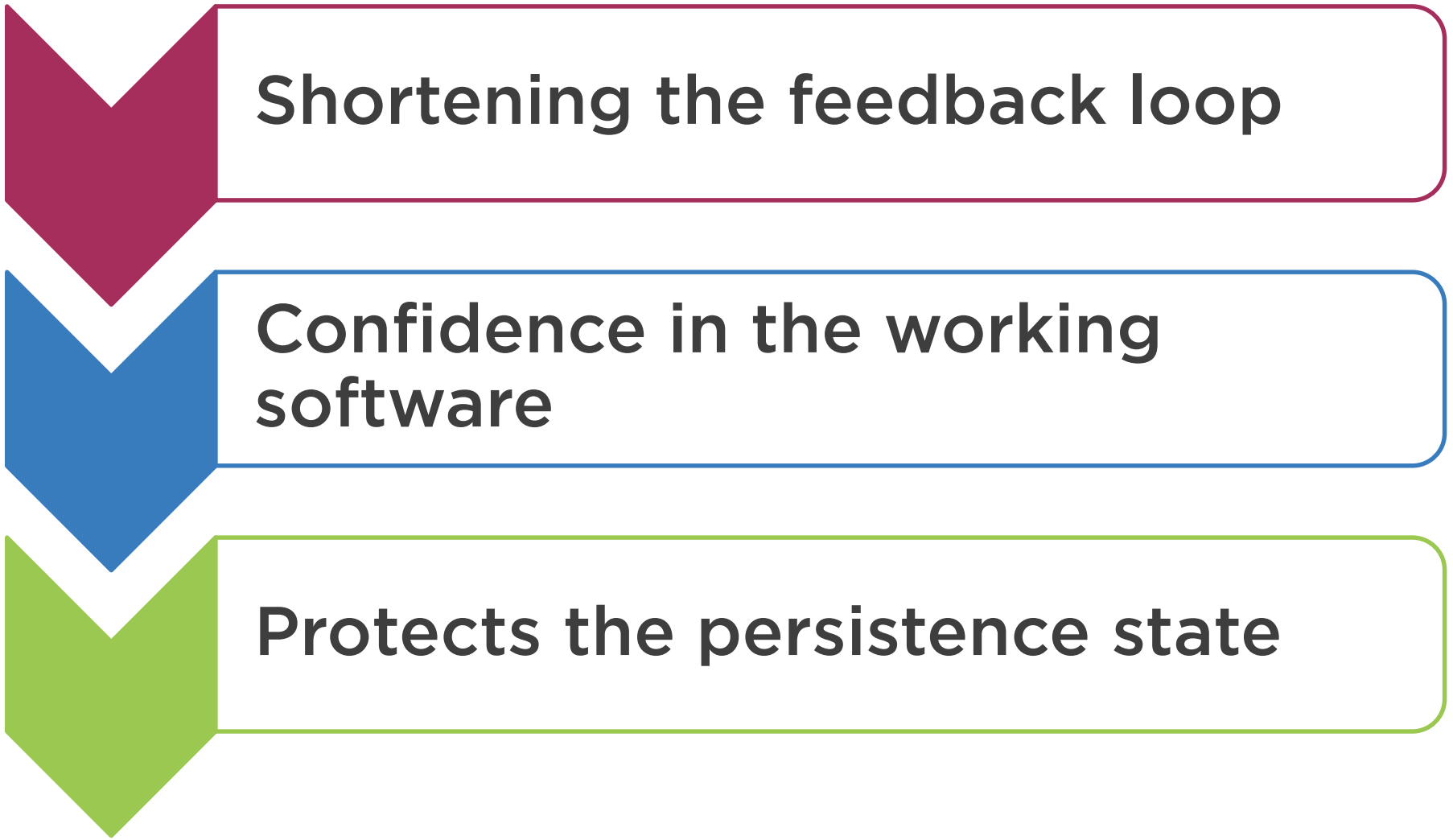
Fail Fast

```
public void ProcessItems(List<Item> items)
{
    foreach (Item item in items)
    {
        Process(item);
    }
}
```

How to React to a Failure



Fail Fast Principle



Where to Catch Exceptions

Where to catch exceptions

```
public static void Main()
{
    try
    {
        StartApplication();
    }
    catch (Exception ex)
    {
        LogException(ex);
        ShowGenericApology();
        Environment.FailFast(null);
    }
}
```

Log exception details

Shut the operation down

Don’t put any domain logic here

Where to Catch Exceptions

3rd party library

Your code

Exceptional situation

Non-exceptional situation

- Should be caught at the lowest level possible
- Catch only exceptions you know how to handle

Where to Catch Exceptions

```
public void CreateCustomer(string name) {
    Customer customer = new Customer(name);
    bool result = SaveCustomer(customer);

    if (!result) {
        MessageBox.Show("Error connecting to the database. Please try again later.");
    }
}

private bool SaveCustomer(Customer customer) {
    try
    {
        using (MyContext context = new MyContext()) {
            context.Customers.Add(customer);
            context.SaveChanges();
        }
        return true;
    }
    catch (DbUpdateException ex) {
        return false;
    }
}
```

Where to Catch Exceptions

```
public void CreateCustomer(string name) {
    Customer customer = new Customer(name);
    bool result = SaveCustomer(customer);

    if (!result) {
        MessageBox.Show("Error connecting to the database. Please try again later.");
    }
}

private bool SaveCustomer(Customer customer) {
    try
    {
        using (MyContext context = new MyContext()) {
            context.Customers.Add(customer);
            context.SaveChanges();
        }
        return true;
    }
    catch (DbUpdateException ex) {
        if (ex.Message == "Unable to open the DB connection")
            return false;
        else
            throw;
    }
}
```

Use return values to define an expected failure

How to distinguish failure reasons?

```
private bool SaveCustomer(Customer customer)
{
    try
    {
        using (MyContext context = new MyContext())
        {
            context.Customers.Add(customer);
            context.SaveChanges();
        }
        return true;
    }
    catch (DbUpdateException ex)
    {
        if (ex.Message == "Unable to open the DB connection")
            return false;

        if (ex.Message.Contains("IX_Customer_Name"))
            return false;

        throw;
    }
}
```

```
private string SaveCustomer(Customer customer)
{
    try
    {
        using (MyContext context = new MyContext())
        {
            context.Customers.Add(customer);
            context.SaveChanges();
        }
        return string.Empty;
    }
    catch (DbUpdateException ex)
    {
        if (ex.Message == "Unable to open the DB connection")
            return "Database is off-line";

        if (ex.Message.Contains("IX_Customer_Name"))
            return "Customer with such a name already exists";

        throw;
    }
}
```

```
private ??? GetCustomer(int id)
{
    try
    {
        using (MyContext context = new MyContext())
        {
            return context.Customers.Single(x => x.Id == id);
        }
    }
    catch (DbUpdateException ex)
    {
        if (ex.Message == "Unable to open the DB connection")
            return ???;
        else
            throw;
    }
}
```

```
private string GetCustomer(int id, out Customer customer)
{
    try
    {
        using (MyContext context = new MyContext())
        {
            customer = context.Customers.Single(x => x.Id == id);
        }
    }
    catch (DbUpdateException ex)
    {
        if (ex.Message == "Unable to open the DB connection")
        {
            customer = null;
            return "Database is off-line";
        }
        else
            throw;
    }
}
```

Recap: The Result Class

- Helps keep methods honest
- Incorporates the result of an operation with its status
- Unified error model
- Only for expected failures

```
private Result SaveCustomer(Customer customer)
{
    try
    {
        using (var context = new MyContext())
        {
            context.Customers.Add(customer);
            context.SaveChanges();
        }
        return Result.Ok();
    }
    catch (DbUpdateException ex)
    {
        if (ex.Message == "Unable to open the DB connection")
            Result.Fail(ErrorType.DatabaseIsOffline);

        if (ex.Message.Contains("IX_Customer_Name"))
            return Result.Fail(ErrorType.CustomerAlreadyExists);

        throw;
    }
}
```

The Result Class and CQS

```
private Result SaveCustomer(Customer customer)
{
    /* ... */
}
```

The Result Class and CQS

```
public void Save(Customer customer)
```

Command

Not excepted to fail

```
public Result Save(Customer customer)
```

Command

Excepted to fail

```
public Customer GetById(long id)
```

Query

Not excepted to fail

```
public Result<Customer> GetById(long id)
```

Query

Excepted to fail

Summary

Exceptions make your code dishonest

Prefer return values over exceptions

Use exceptions for exceptional situations

In the face of failure, stop the current operation entirely

Catch exceptions at:

- Highest level possible for logging purposes
- Lowest level possible to process an exception from a 3rd party library

The Result class helps keep your methods honest

The Result class and the CQS principle

In the Next Module

Primitive Obsession