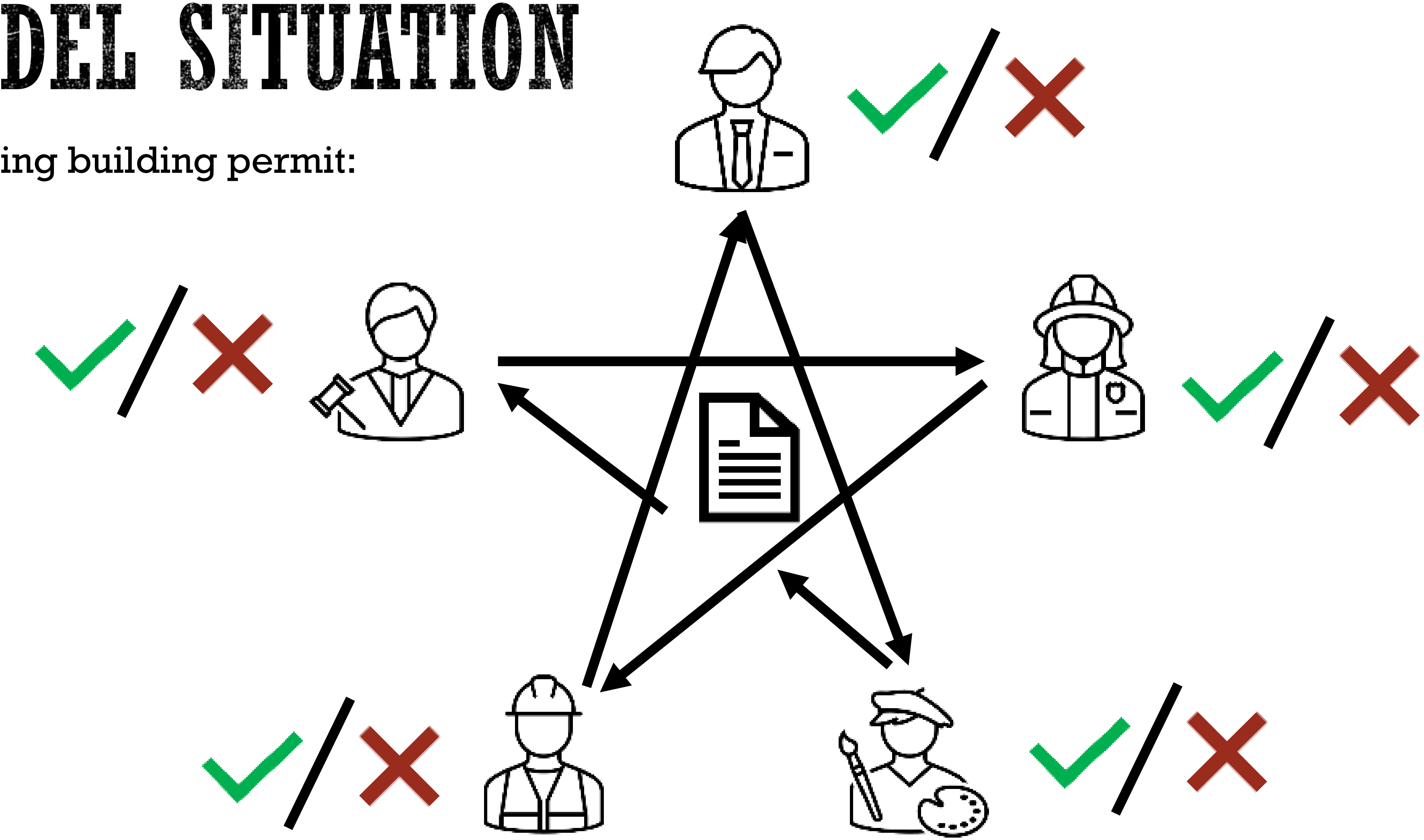
CHAIN OF RESPONSIBILITY

MODEL SITUATION

Obtaining building permit:



SIMPLE IMPLEMENTATION IN C#

```
static bool ApplyForBuildingPermit(Permit permit)
{
    if (!ZoningDepartment.CheckSitePlan(permit))
        return false; // Site plan did not meet zoning regulations
    else if (!BuildingDepartment.CheckBuildingCodes(permit))
        return false; // Building plan did not meet building codes
    else if (!FireDepartment.CheckFireSafety(permit))
        return false; // Fire safety plan did not meet regulations
    else
        return true; // Permit approved
   }
```

SIMPLE IMPLEMENTATION SUMMARY

- The implementation may become hard to maintain and modify as the number of departments and permit requirements grow.
- It violates the Open-Closed Principle as new departments or permit requirements will require modification of the existing code.
- The implementation may result in code duplication or long method chains.
- It may be harder to test due to the tight coupling of the code and logic.
- It is not modifiable during runtime.

OOP IMPLEMENTATION IN C#

```
public abstract class Department
{
    public abstract bool CheckPermit(Permit permit);
}
public class ZoningDepartment : Department
{
    public override bool CheckPermit(Permit permit)
    {
        return 
permit.SitePlan.MeetsZoningRegulations;
    }
}
public class BuildingDepartment : Department
{
    public override bool CheckPermit(Permit permit)
    {
        return 
permit.BuildingPlan.MeetsBuildingCodes;
    }
}
public class FireDepartment : Department
{
    public override bool CheckPermit(Permit permit)
    {
        return 
permit.FireSafetyPlan.MeetsRegulations;
    }
}
```

```
public class PermitApprovalChain
{
    private List<Department> departments = 
                                  new List<Department>();

    public PermitApprovalChain()
    {
        departments.Add(new ZoningDepartment());
        departments.Add(new BuildingDepartment());
        departments.Add(new FireDepartment());
    }

    public bool ApplyForBuildingPermit(Permit permit)
    {
        foreach (Department department in departments)
        {
            if (!department.CheckPermit(permit))
            {
                return false; // Permit denied
            }
        }
        return true; // Permit approved by all
    }
}
```

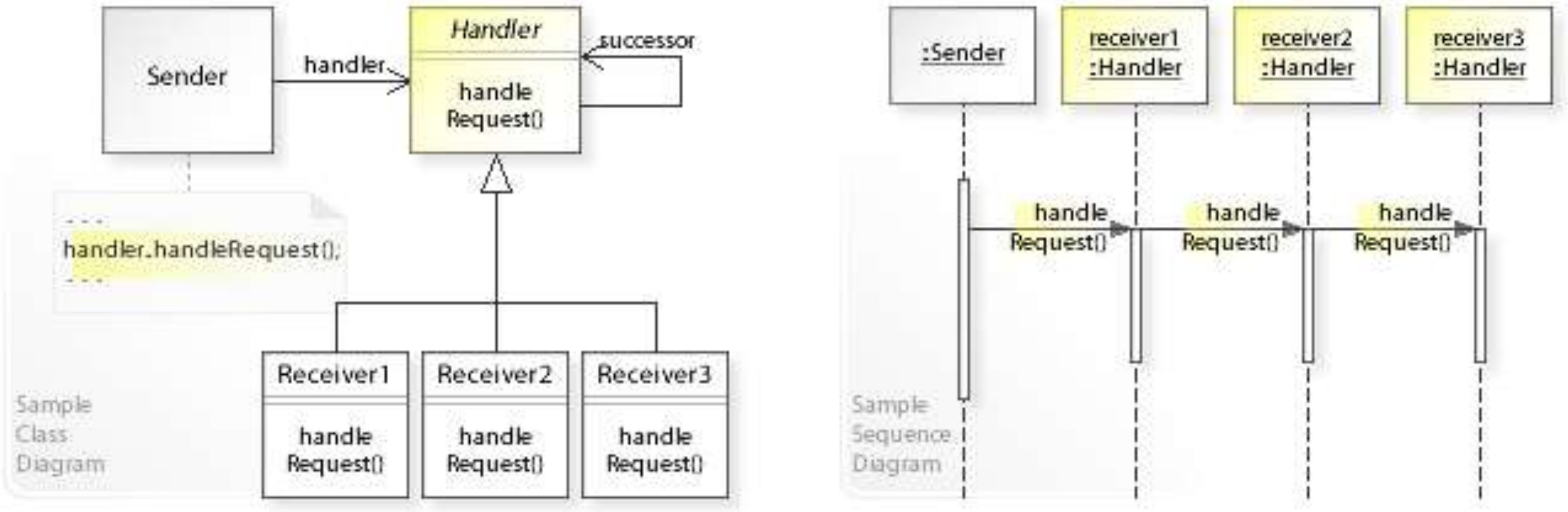
OOP IMPLEMENTATION SUMMARY

- It may result in slower performance due to the overhead of managing a list of objects.
- It may not be flexible enough to handle complex scenarios with multiple decision points or branching logic.
- It may not provide a clear separation of concerns and centralizes point of control.
- Client must keep links to all department objects and must understand the structure of the approval process

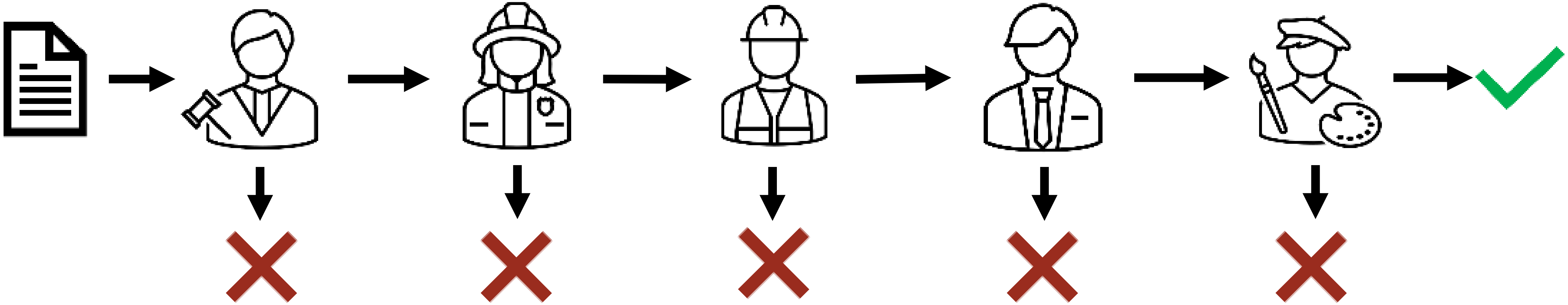
CHAIN OF RESPONSIBILITY INTRODUCTION

- It is **behavioral design** pattern.
- The idea is to create a **chain of handler objects**, where each handler can decide whether to handle a request or pass it to the next handler in the chain.
- The pattern promotes **loose coupling** and **separation of concerns**, as each handler is responsible for a specific task.
- It **provides flexibility and extensibility**, as new handlers can be added or removed from the chain at runtime without affecting the overall structure of the code.
- Commonly used in scenarios where there are multiple objects that can handle a request, or where the handling of a request requires a series of steps or decision points.

GENERAL SCHEMA OF THE PATTERN



SOLUTION TO MODEL PROBLEM



FINAL IMPLEMENTATION IN C#

```
public abstract class Department
{
    protected Department successor;

    public void SetSuccessor(Department successor)
    {
        this.successor = successor;
    }

    public abstract bool CheckPermit(Permit permit);
}
public class ZoningDepartment : Department
{
    public override bool CheckPermit(Permit permit)
    {
        if (permit.SitePlan.MeetsZoningRegulations)
        {
            return
              successor != null ? successor.CheckPermit(permit) :
true;
        }
        return false;
    }
}
```

```
public class PermitApprovalChain
{
    private Department entryPoint;

    public PermitApprovalChain()
    {
       entryPoint = new ZoningDepartment();

       var buildingDepartment =
                         new BuildingDepartment();
       var fireDepartment =
                         new FireDepartment();

       entryPoint.SetSuccessor(buildingDepartment);

buildingDepartment.SetSuccessor(fireDepartment);
    }

    public bool ApplyForBuildingPermit(Permit permit)
    {
       return entryPoint.CheckPermit(permit);
    }
}
```

REAL-LIFE EXAMPLES

- **Web applications:** The Chain of Responsibility pattern is often used to handle requests through a middleware chain, where each middleware component can perform specific tasks or validations before passing the request to the next component. (Express.js middleware in Node.js, ASP.NET middleware in .NET).
- **GUI toolkits:** The pattern is used to handle user input events and delegate them to the appropriate UI component or event handler. (Java Swing, WinForms)
- **Logging frameworks:** The pattern is used to handle logging requests and pass them through a chain of loggers.
- **Exception handling:** The pattern is used to handle exceptions through a chain of handlers with increasing levels of abstraction or specificity, such as catching exceptions at the application level, then at the module level, and so on.

RELATION TO COMPOSITE

- Both involve creating hierarchies of objects and delegating responsibilities.
- They differ in their intent and implementation.
- In some cases, the Composite and Chain of Responsibility patterns can be combined to create complex hierarchical structures

THANK YOU FOR YOUR ATTENTION