welcome

Denver Dev Day - June 23, 2017

Microsoft

TEKsystems
Our people make IT possible.

revgen partners
DRIVING PROFITABILITY®

PERFICIENT

KidsTek
Reach Higher through Technology Education

Better Business Logic with Typescript

Implementing and Using Design Patterns with Angular

Contact Information

- Matt Vaughn
  - E: matt.vaughn@buildmotion.com
  - W: www.buildmotion.com
  - GitHub: https://github.com/buildmotion
- Code Samples and Presentation
  - GitHub: https://github.com/buildmotion/better-business-logic-with-typescript

Objective

- The session will show you how to use and implement design patterns (like Composite, Builder, Template Method and others) using Typescript. The session will demonstrate code reuse strategies and extensibility techniques for building robust Angular Services and Business Logic.
- The session will demonstrate how to implement shared Angular services and how to use them in Angular domain services. The services will include rich business logic layers that use the angular-rules-engine and angular-actions NPM packages.

Business Logic

Business Logic – Why important?

- Heart of the application.
- Defines the business domain.
- Domain specific algorithms, intellectual property, etc.

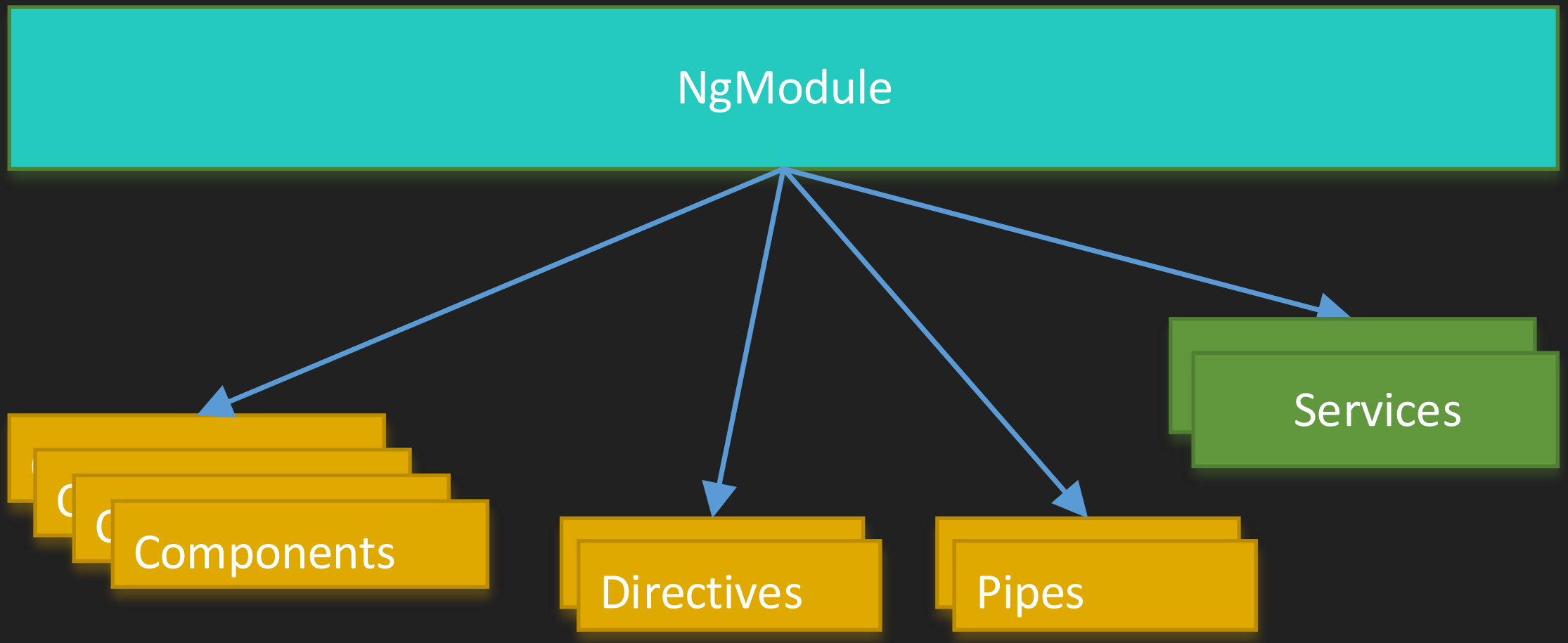
Business Logic – Why Difficult?

- Inconsistent implementation causes problems with extensibility/maintainability.
- Architecture, design, or implementation makes it difficult to test.
- Different implementation styles by different developers create inconsistent code base.
- Single Responsibility Principle not followed - parts of code do too much.
- Separation of Concerns Principle not followed - parts of the code cross boundaries. BL contained in different layers of the application (UI, services, BL layer, database).
- Lack of structure or use of defined patterns.

Angular Modules

What is an Angular Module?

- What is a module…?
  - A container of related elements.
  - A module may have: components, directives, pipes, or services.
  - Angular application can only one root module: AppModule.
  - Angular application can have zero-to-many feature modules.
    - Core
    - Shared
    - Feature


NgModule
Components
Directives
Pipes
Services

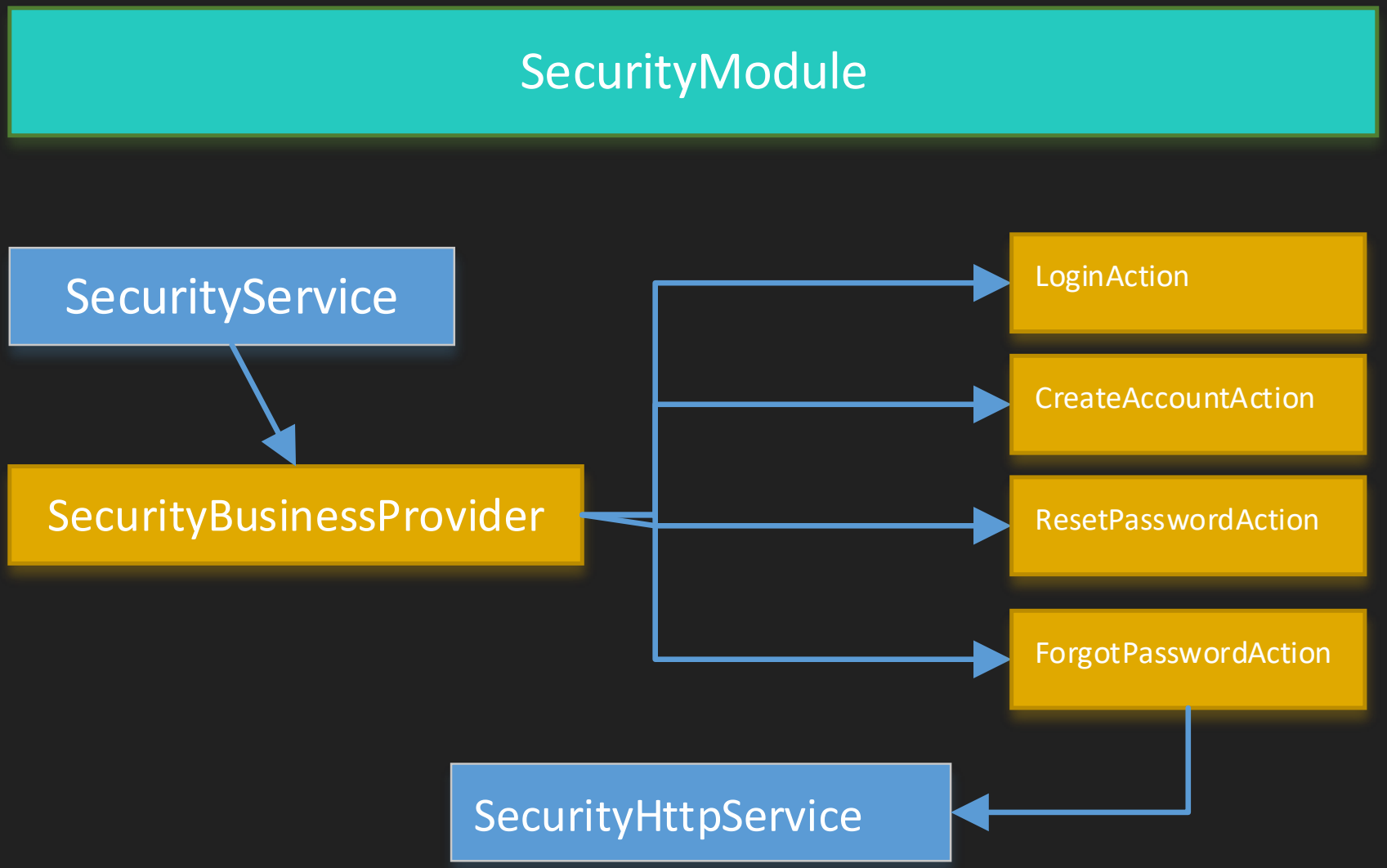
Are there any limits…really?

- How many modules can you Angular application have?
- Must a module have a service?
- Must a module have components?
- Are there different kinds of modules?
- What kind of modules does my application need?

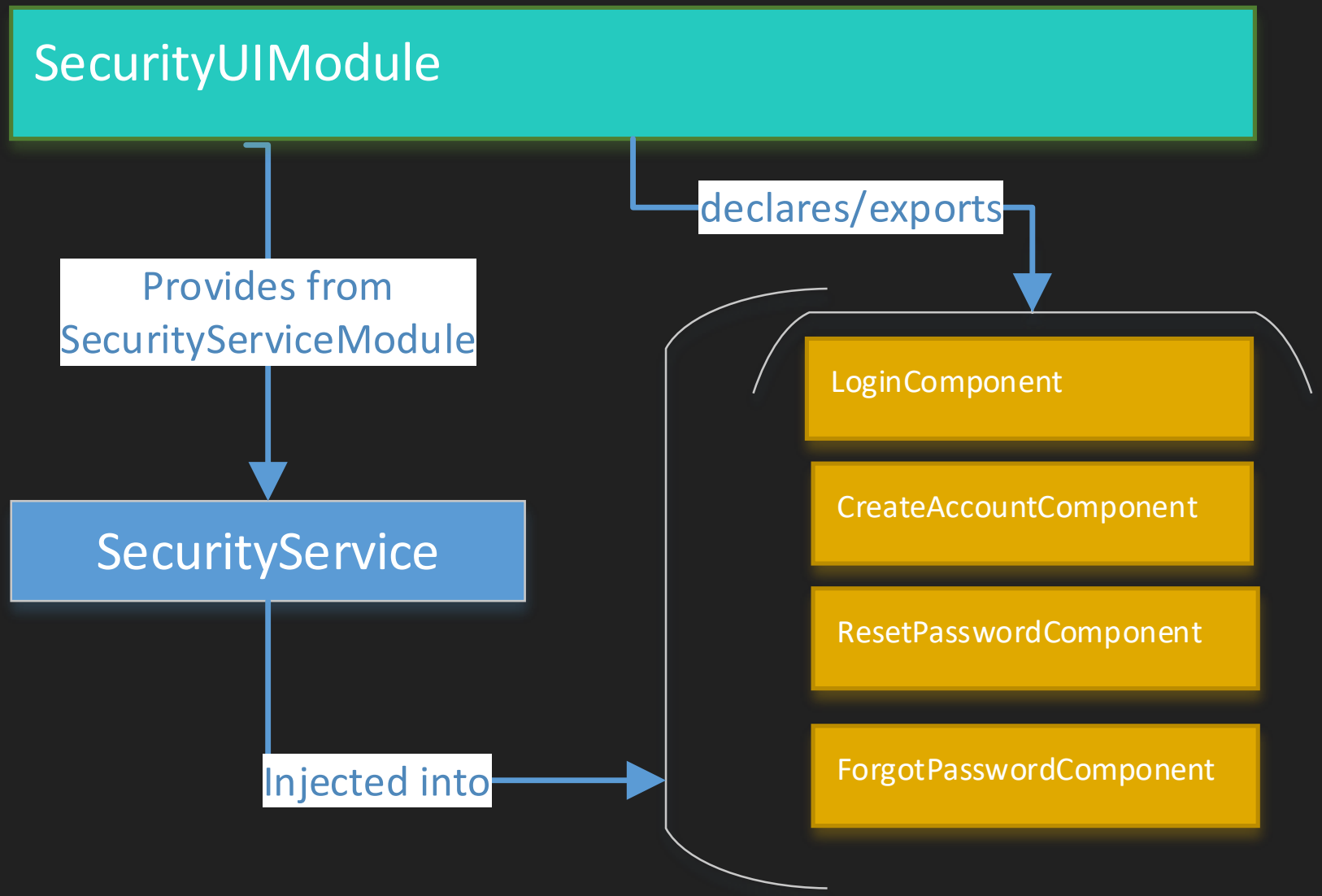
Consider Module…Categories/Types

- **Shared**
  - Angular: NgModule, CommonModule, HttpModule, FormsModule, etc.
  - Third-Party Modules: Wijmo, Material Design
- **Core**: Application-Level Modules
  - Modules: `PagesModule, LayoutModule`
  - Services: `LoggingService`
  - Components: `MenuComponent, FooterComponent`
- **Infrastructure:** Base/Foundation
  - Base and Framework (i..e, buildmotion-foundation, angular-actions, angular-rules-engine)
- **Domain Service**: Service-Only
  - Module: `SecurityModule`
  - Service: `SecurityService`
  - Components: *none*
- **Domain UI**: UI-Only
  - Module: `SecurityUIModule`
  - RoutingModule: `SecurityRoutingModule`
  - Components: `LoginComponent, SignUpComponent. ResetPassword, ForgotPassword`

Feature Module :: SecurityModule



Feature Module :: SecurityUIModule



Design Patterns

A brief introduction

What is a pattern?

- Is a general reusable solution to a commonly occurring problem within a given context in software design.
- Design patterns can speed up the development process by providing tested, proven development paradigms.
- Improves code readability for coders and architects who are familiar with the patterns.
- Many patterns based on Object Oriented Programming techniques: inheritance, abstraction, encapsulation, and polymorphism.
- Many patterns support and promote:
  - Single Responsibility
  - Separation of Concerns
  - S.O.L.I.D Principles

Why Use Design Patterns?

- What does this do for me or my team?
  - Creates a consistent code base for improved maintainability.
  - Inherently allows for more extensibility points.
  - Promotes a more testable solution (unit and integration tests) with mocks and fakes. Better quality.
  - Allows dependency injection opportunities.
- What problems to they solve?
  - Help with refactoring code to improve testability, extensibility, and maintainability.
  - Many patterns allow for additional extensibility points.
  - Use existing patterns without creating atypical solutions for common problems.
  - The team has a recipe for implementing parts of the application.

Patterns are Everywhere!

- Many examples of patterns in real life.
- Software and Frameworks
  - .NET
  - ASP.NET
  - ASP.NET MVC
  - Angular
- Typescript…

“I see Angular Patterns…”

- Template Method
  - Lifecycle Hooks: ngOnChanges, ngOnInit, ngDoCheck, and ngOnDestroy
- Composite
  - A component can reference a component that has a component that has a component.
- Mediator
  - A parent-component or injected service can mediate communication between loosely-coupled components/child-components.
- Observer
  - A component can subscribe to and react to an Observable response.
- Facades
  - Services provide end-points to perform some action. How the action is performed is not known by the calling component.
- Dependency Injection (Inversion of Control – IoC)
  - Services are instantiated and injected into components.

“Use the Angular…Luke.”

- Use the elements in the Angular Framework
  - Modules
  - Services
  - Components
  - Dependency Injection
    - Code Smell: `var thingService = new ThingService(..);`
- Use the “force” – Use the “angular”.
  - If there is a way to do it in Angular, learn and use that mechanism.

Façade Pattern

Façade Pattern Details

- Definition: A single class that represents an entire subsystem.
  - Useful in API design
  - Useful in Service Oriented Architectures (SOA)
  - Typically implements an *Interface*.
  - Typically does not provide an implementation – but, delegates to a lower-level business layer.
- An Angular Service can be a *Façade*.

Template Method Pattern

There is a method to this madness.

Business Logic Madness

- Business Logic
- Data Input Validation
- Business Rule Implementation
- Authorization or Permissions?
- Retrieving and persisting data
- Handling exceptions.
- Handling error responses from the backend.
- Notifications to Users

Business logic goals?

- Goals:
  - Consistent
  - Maintainable
  - Testable
  - Extensible
  - High Quality
- How can we achieve this?

Elevate Your Business Logic

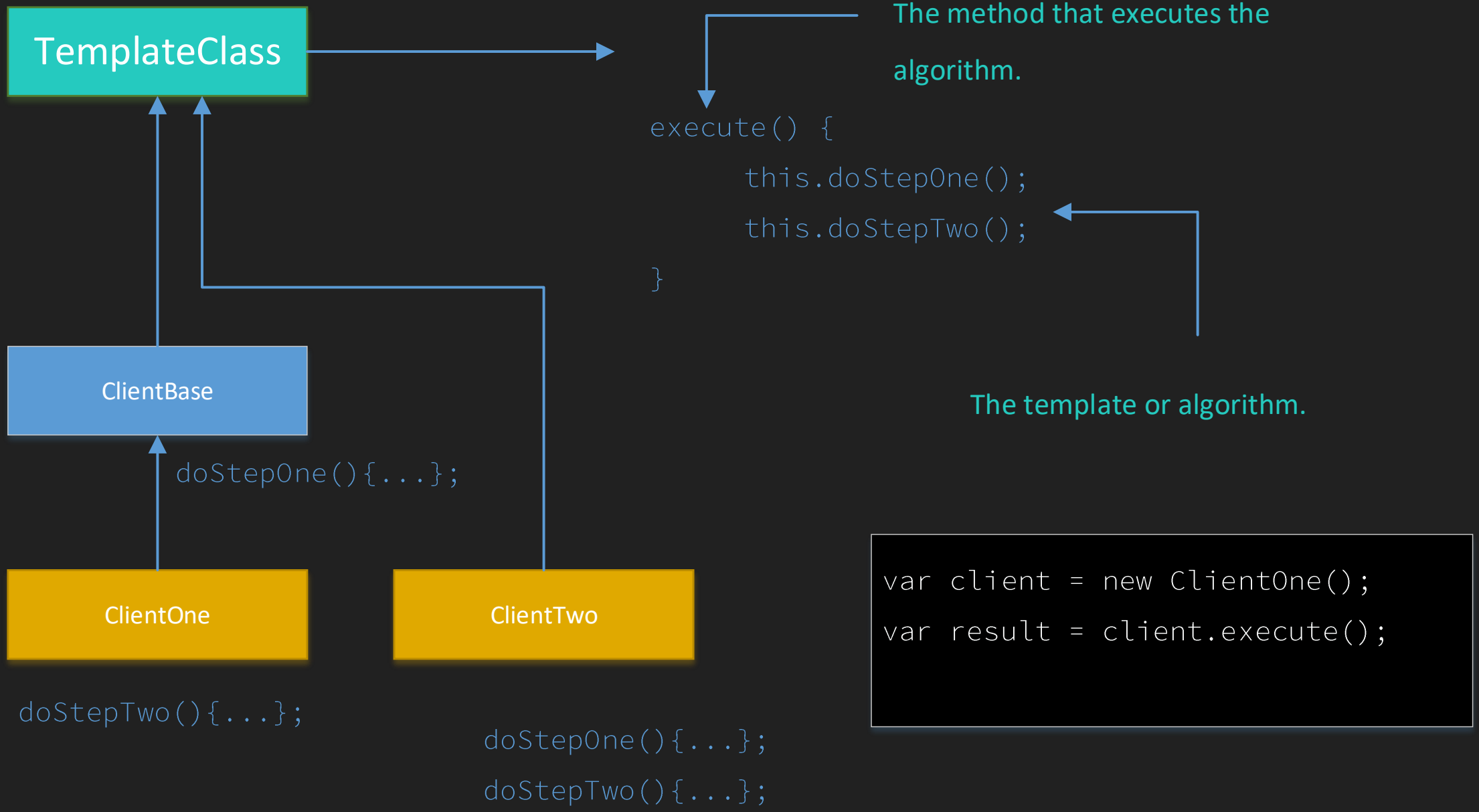
- Stop the Method Madness
  - Use classes to implement your business logic.
  - A single class with a single responsibility, a single unit of work.
  - This is not the typical Manager class with a bunch of methods chained together.
- Class Advantage
  - Can use one or more `Constructors`.
  - Can have base classes - `Inheritance`.
  - Can implement an `Interface`.
  - Can expose typed `Properties`.
  - Can encapsulate members using `private`.
- Design Patterns
  - Template Method (Lifecycle/Pipeline)
- 100% Testable

Template Method Pattern

- Where have you seen this one?
  - ASP.NET Forms circa 2001: PreInit, Init, InitComplete, PreLoad, Load, LoadComplete, PreRender, etc.
  - Angular 2: ngOnChanges, ngOnit, ngDoCheck, ngOnDestroy
- If it is good enough for managing components and web forms with millions of different implementations – but, all using the same algorithm.

Template Method Pattern :: Details

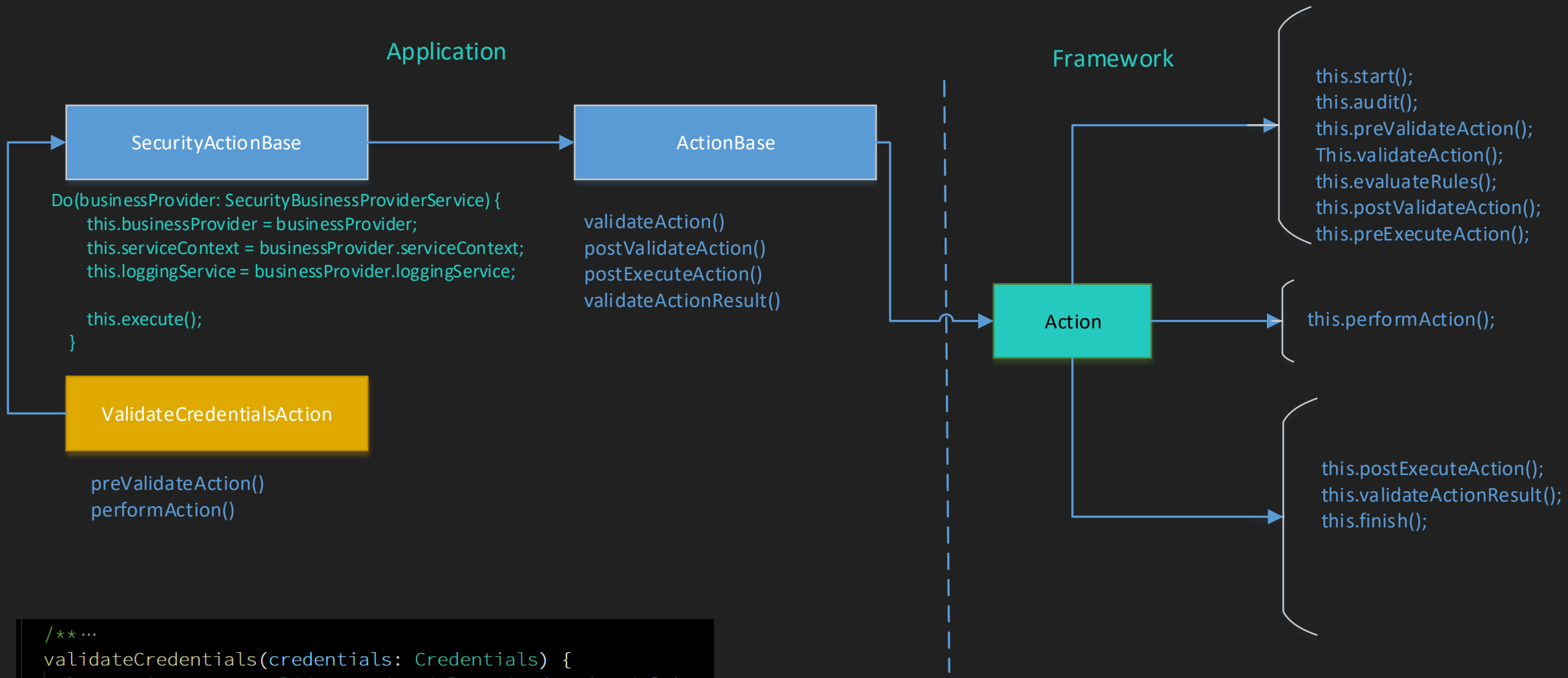
- **Definition**: defines an algorithm (i.e., pipeline of methods and/or events) that can be implemented with concrete classes without changing the structure of the algorithm.
- Creates a consistent well-defined flow – that also enables default or shared behaviors to be implemented for all implementing classes.
- Allows for customized implementations of pipeline items within the algorithm.
- Allows for extensibility end points without disturbing the structure of the algorithm.
- Simplifies the target concrete implementation by allowing other non-essential details to be handled by the base/abstract classes that define and implement the template algorithm.
- Example: GitHub


TemplateClass
The method that executes the algorithm.
execute() {
this.doStepOne();
this.doStepTwo();
}
ClientBase
The template or algorithm.
doStepOne(){...};
ClientOne
ClientTwo
doStepTwo(){...};
doStepOne(){...};
doStepTwo(){...};
var client = new ClientOne();
var result = client.execute();

DEMO :: Taco Template

DEMO

- http://dev.hybridmobileapp.net




```
  /**…
  validateCredentials(credentials: Credentials) {
    let action = new ValidateCredentialsAction(credentials);
    action.Do(this);
    return action.response;
  }
}
```

Action Pipeline

1. Service delegates to the Business Layer to process request.
2. The Business Layer initializes a new Action and Executes action.
3. The concrete action implements the `preValidationAction()` method and the `performAction()` method.

1

```
createUserAccount(userAccount: UserAccount) {
  this.resetServiceMessages();
  this.loggingService.log(this.serviceName, Severity.Information,
  return this.businessProvider.createUserAccount(userAccount);
}
```

2

```
createUserAccount(userAccount: UserAccount) {
  let action = new CreateUserAccountAction(userAccount);
  action.Do(this);
  return action.response;
}
```

3

```
export class CreateUserAccountAction extends SecurityActionBase {

    response: Observable<Response>;
    httpBase: HttpBaseService;

    constructor(private userAccount: UserAccount) {…
    }

    /**…
    preValidateAction() {…
    }

    /**…
    performAction() {…
    }
}
```

Template Method Recap

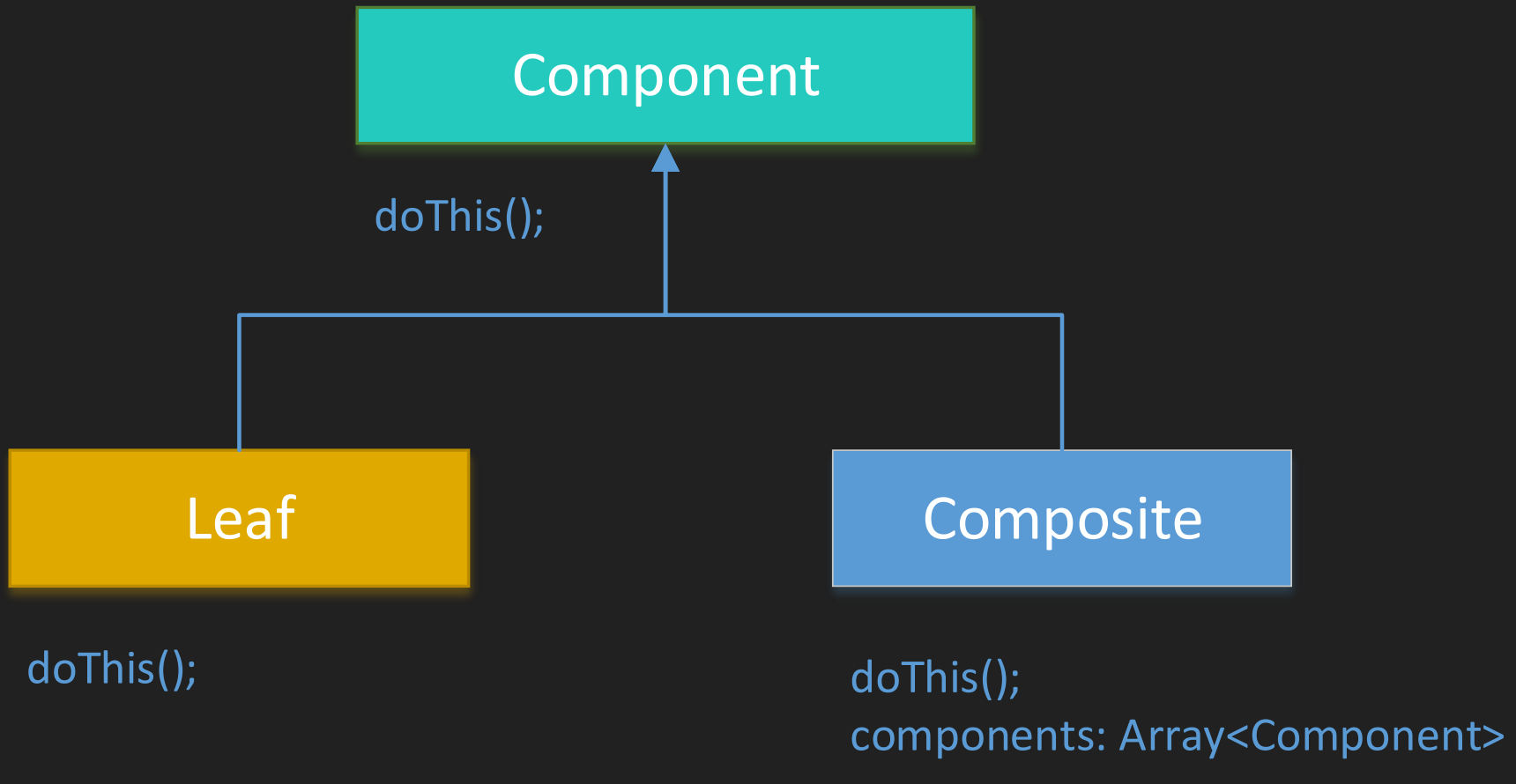
- You can define the algorithm/recipe for your business logic processing.
  - Logging
  - Auditing
  - Authorization
  - Validation and Business Rule processing
- Consistent and Reliable
- Provides extensibility and customization opportunities.
- Uses Object Oriented Programming
- Scaffold with Templates

Composite Pattern

Composite Pattern

- Where have you seen this one?
  - File Explorer
  - Angular 2 Components

Composite Pattern



Demo(s)

- Angular Sample Application
  - Rules
    - Simple (Primitive)
    - Complex (Composite)
- Account Signup

Composite Pattern Recap

- Simple to implement.
- Create simple/complex items in a hierarchy or tree-like view.
- Process/Handle either (simple/composite) types consistently.

Conclusions

Final thoughts…

- Business Logic is very important – yet challenging.
  - Needs to be protected as a valuable resource.
- Patterns and Frameworks help control the chaos.
  - Understand and use the elements that Angular provides more fully.
- Higher Quality Systems
  - Testable
  - Extensible
  - Maintainable
- Take time to learn and understand different design patterns.

Web References

- GoF Design patterns in .NET/C#
- Design Patterns General
  - Creational
  - Structural
  - Behavioral
- Principles:
  - Single Responsibility
  - Separation of Concerns
  - S.O.L.I.D Principles

Book References

- Design Patterns : Elements of Reusable Object-Oriented Software
  - ISBN-13: 978-0201633610, ISBN-10: 0201633612
- Head First Design Patterns
  - ISBN-13: 978-0596007126, ISBN-10: 0596007124
- Clean Code
  - ISBN-13: 978-0132350884, ISBN-10: 0132350882
- The Pragmatic Programmer
  - ISBN-13: 978-0201616224, ISBN-10: 020161622X
- Patterns of Enterprise Application Architecture
  - ISBN-13: 978-0321127426, ISBN-10: 0321127420

Angular Resources

- https://angular.io:
- https://rangle.io:
  - Rangle's Angular 2 Training Book
  - Functional Forms
- Become a Ninja with Angular: sample chapters
- Tour of Heroes Application with Multiple Modules

Miscellaneous

- “Good programmers write good code. Great programmers steal.” - Unknown
  - Learn, Use, Copy, Borrow, and/or Steal.