Design Principles Using OpenEdge

Mauricio dos Santos
Principal Consultant
Progress Software Corp.
Professional Services

Audience for this course

This course is for experienced ABL developers who have a good understanding of Object Oriented development and want to achieve a higher level of software quality from design to development, maintenance and deployment.

Course prerequisites

Before you begin this course, you should have experience with:

- Basic Object Oriented understanding from a developer's point of view
- Writing ABL code to:
  - Create simple abstract classes and interfaces
  - Create simple concrete classes
- Developer Studio for OpenEdge to create projects, procedures, classes and edit ABL code.
- Progress Application server for OpenEdge (PAS for OpenEdge)
  - Modifying a launch configuration
  - Starting and stopping a PAS for OpenEdge instance

Introduce yourself

- Your name and your job.
- Name of your company and its type of business.
- Your technical background.
- Any prior experience with Progress Software products?
- What would you like to learn from this course?

Learning objectives for this course

After taking this class, you should be able to:

- Understand and explain each design principle.
- Have a more profound and concise understanding of key object oriented concepts.

System and software requirements for this course

- Hardware/platform
  - Windows 64 bit
  - 2 Gb RAM
  - 100 Mb disk space for course files
  - 6 Gb disk space for Progress OpenEdge install files and installation
- Progress Developer Studio for OpenEdge 11.7 or later
  - Must install 64 bit version of Developer Studio with one of:
    - Developer Studio for OpenEdge license
    - OpenEdge Developer Kit (OEDK) Classroom Edition (no license required)
- See **ExerciseSetup** for instructions to follow before you begin the exercises of this course

Agenda

- Separation of Concerns (SoC)
- SOLID Principles
- Package Cohesion Principles
- Package Coupling Principles
- Commonly Referenced Principles
- Naming Conventions

Separation of Concerns

- ‘Invented’ in 1974, not unique to OO
- Modularity
- Encapsulation / Information hiding
- SOA
- Micro Services

SOLID Principles

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

- A class should have only one reason to change.
- So a responsibility is a family of functions that serves one particular actor.
- An actor for a responsibility is the single source of change for that responsibility.

Open-Closed Principle (OCP)

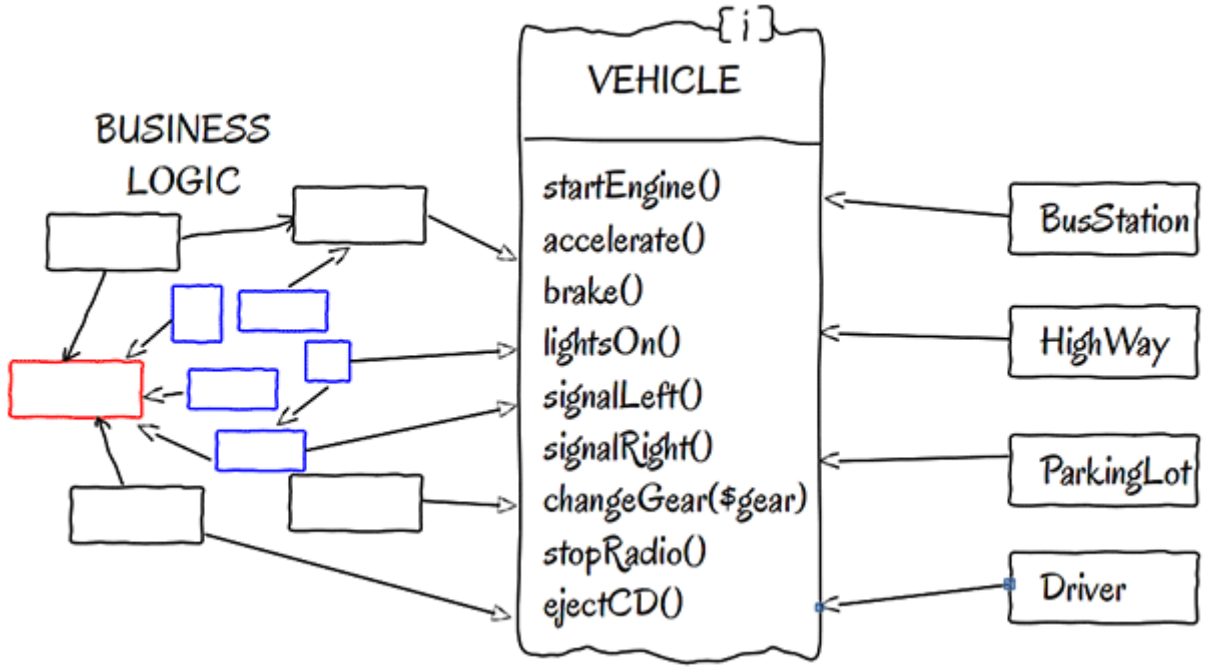
- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- Design modules, classes and functions in a way that when a new functionality is needed, we should not modify our existing code but rather write new code that will be used by existing code.

Liskov Substitution Principle (LSP)

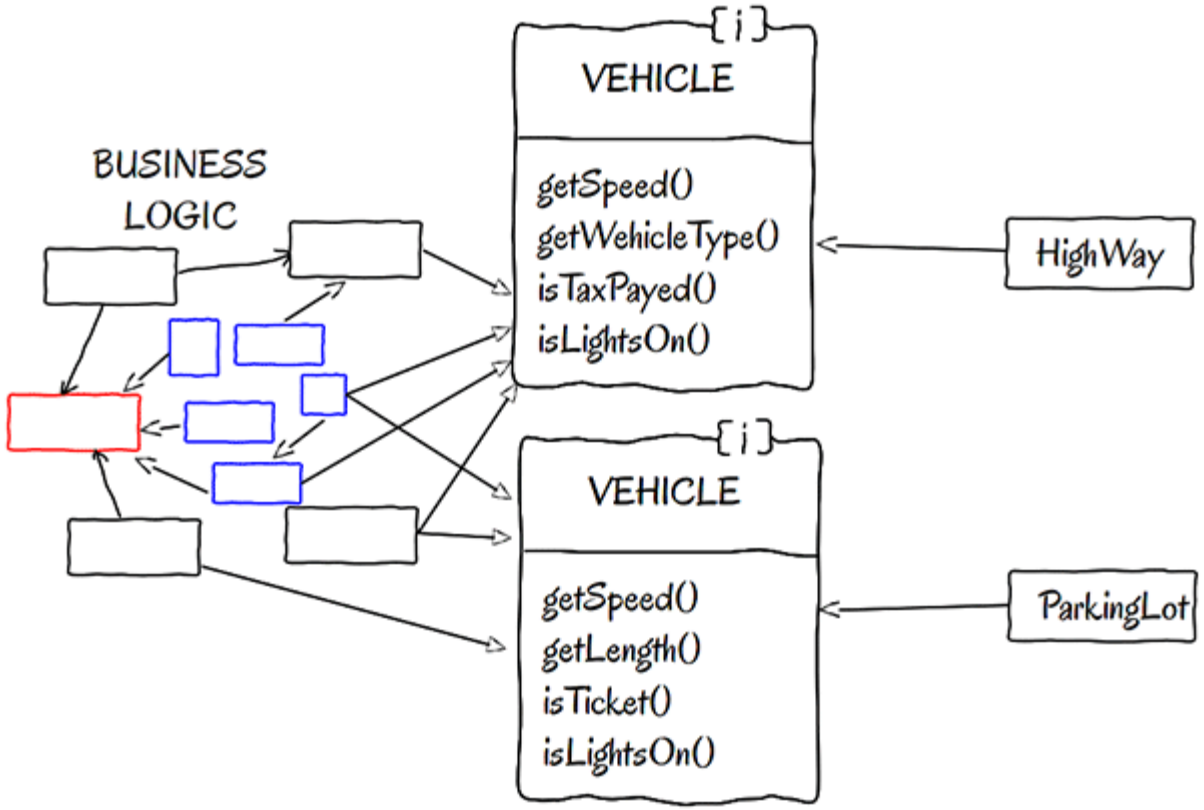
- Subtypes must be substitutable for their base types
- S <: T

Interface Segregation Principle 1/2 (ISP)

- No client should be forced to depend on methods it does not use
- Do not design interfaces bigger than necessary
- Split large interfaces into smaller ones so clients only need to know about the methods they are interested in



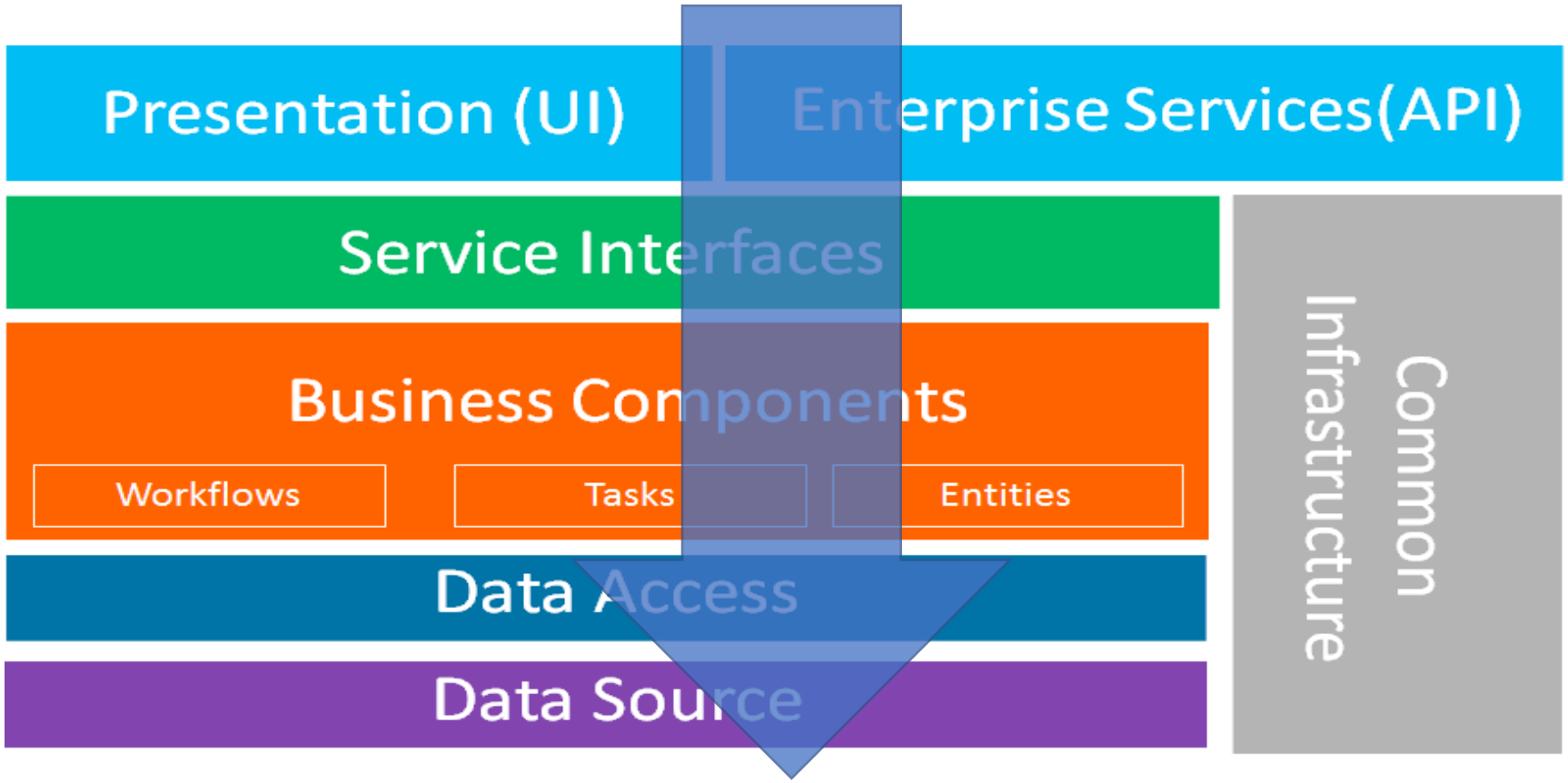
Interface Segregation Principle 2/2 (ISP)



Dependency Inversion Principle 1/2 (DIP)

- High-level modules should not depend on low-level modules — both should depend on abstractions
- Abstractions should not depend upon details
- Details should depend upon abstractions

Dependency Inversion Principle 2/2 (DIP)



Package Cohesion Principles

- Reuse/Release Equivalence Principle (REP)
- Common Closure Principle (CCP)
- Common Reuse Principle (CRP)

Reuse/Release Equivalence Principle (REP)

- The granule of reuse is the granule of release
- Only components that are released through a tracking system can be effectively reused
- This granule is the package
- Use Semantic Versioning
  - MAJOR version when you make incompatible API changes
  - MINOR version when you add functionality in a backwards-compatible manner
  - PATCH version when you make backwards-compatible bug fixes

Common Closure Principle (CCP)

- The classes in a package should be closed together against the same kinds of changes
- A change that affects a package affects all the classes in that package and no other packages
- “The Single Responsibility Principle for packages”
- How does your package structure look like? Horizontal or Vertical? Which packages are affected by a change?

Common Reuse Principle (CRP)

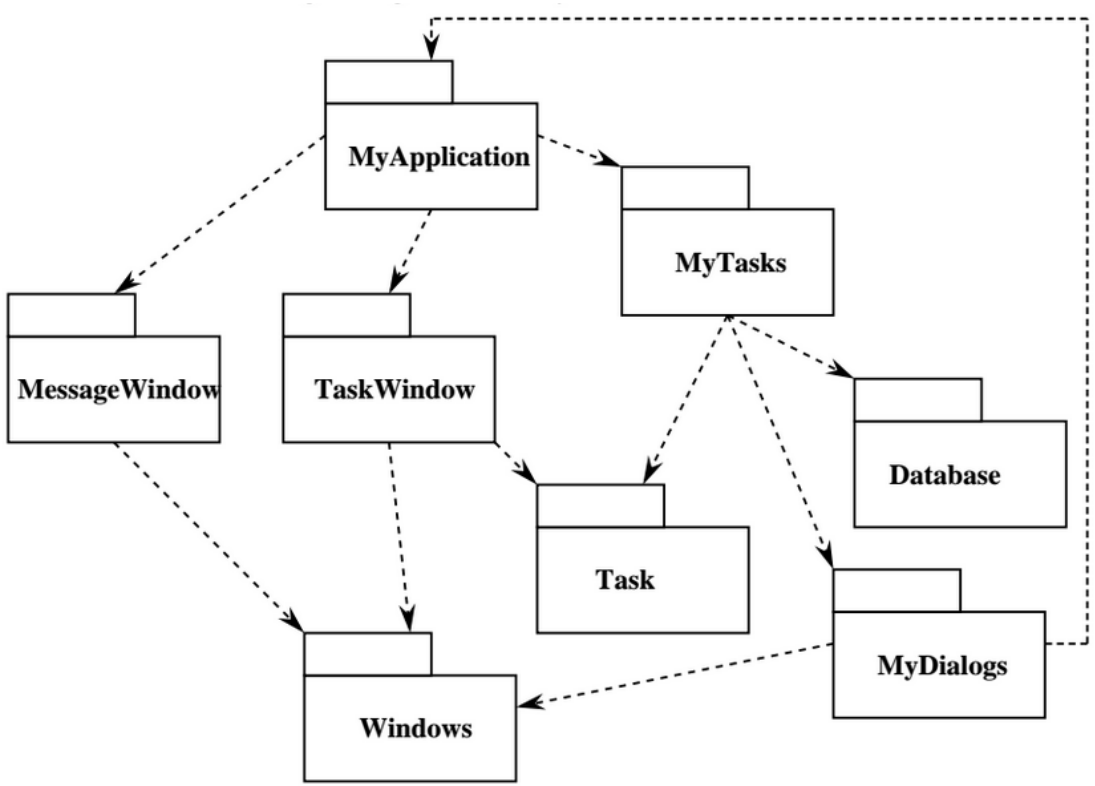
- The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all
- “If a package is being released because of changes to a class that I don’t care about, then I will not be very happy about having to revalidate my application”

Package Coupling Principles

- The Acyclic Dependencies Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)

The Acyclic Dependencies Principle (ADP)

- The dependency structure between packages must be a Directed Acyclic Graph (DAG)
- There must be no cycles in the dependency structure

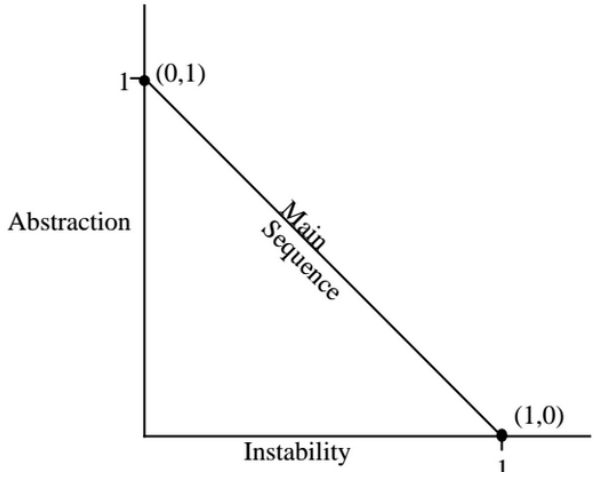


Stable Dependencies Principle (SDP)

- The dependencies between packages should be in the direction of increasing stability, i.e., a package should always be dependent on a package that is more stable than it is

Stable Abstractions Principle (SAP)

- Packages that are maximally stable should be maximally abstract
- Unstable packages should be concrete
- The abstraction of a package should be in proportion to its stability
- The more stable the classes are, the more of them should become abstract
- A = abstractClasses ÷ totalClasses



Commonly Referenced Principles

- Program to an Interface
- Favor Composition over Inheritance

Program to an Interface

- not an implementation
- Disadvantage: if you add methods to the interface, all clients need to adapt
- Abstract classes don’t have this disadvantage

Favor Composition over Inheritance

- When some behavior of a domain object can change with other features remaining the same
- Changes in the interface of a super class can break code that uses subclasses
- Make sure inheritance models the is-a relationship
- Is an Employee a Person?

Naming Conventions 1/3

- Use Intention-Revealing Names
  - Not lOk but lCreditApproved
- Avoid Disinformation
  - lNotNegative
  - iTotal and iTotals
- Make meaningful Distinctions
  - ProductData and ProductInfo
- Use Pronounceable Names
  - genymdhms -> generationTimestamp
- Avoid Encodings
  - cAddress -> Address
  - iBusinessEntity -> BusinessEntity

Naming Conventions 2/3

- Avoid Mental Mapping
  - makeInvoiceNegative -> makeCreditNote
  - Class names
  - Customer, Account, AddressParser
- Method names
  - deletePage, save
- Don’t be cute
  - holyHandGrenade -> deleteItems
- Pick one word per concept
  - fetch, retrieve, get…

Naming Conventions 3/3

- Don’t pun
  - Don’t use add in different classes if you’re not really adding
- Use solution domain names
  - JobQueue
- Use problem domain names
  - EntitiesRelation -> ProductWithCategory
- Use meaningful names in their self context
  - addressCity, addressHomeNumber, addressPostCode -> city, homeNumber, postCode

Exercises

- **Exercise 1** will ask you to simply create a class with a method — but does that method belong there?
- **Exercise 2** will then make the application more reusable, maintainable and extensible by creating an interface and separate classes for printing.
- **Exercise 3** again will ask you to simply create a class with a couple of methods — but again are they best implemented there?
- **Exercise 4** will provide the solution to achieve a higher level of reusability, maintainability and extensibility. Even though we have not learned about design patterns yet, this exercise will demonstrate the Template Method pattern which is a type of Behavioral pattern. As its name implies, this pattern revolves around providing a “template” for a given intended (expected) behavior, therefore ensuring an object can function in a standardized yet configurable way (based on its client’s needs).

Exercise 1

- Create a new OpenEdge general project and new folder/package named Exercise_01
- Create a class named Book
- Add 2 public properties:
  - Title
  - Author
- Add an empty method turnPage
- Add a method named printCurrentPage with this (logic) content below:
  - `MESSAGE "Once upon a time…" VIEW-AS ALERT-BOX.`

Exercise 2

- Create a new folder/package named Exercise_02
- Copy class Book from previous exercise
- Create an interface named Printer
- Make 2 implementations (i.e. create two concrete classes):
  - PrintMessage.cls to print book's current page using MESSAGE statement
  - PrintLog.cls to print book's current page using LOG-MANAGER
- Change Book class so it does not print anything but rather return the book's current page contents
- Create a test procedure (client) to instantiate a Book and a desired Print type object in order to print that book's current page

Exercise 3

- Create a new folder/package named Exercise_03
- Create a class named Car
- Add two properties:
  - Brand
  - Model
- Add a method to calculate fuel consumption
- Add a method to calculate remaining mileage

Exercise 4

- Create a new folder/package named Exercise_04
- Copy the Car class from previous exercise
- A Car can be either Hybrid or Gasoline; calculation of fuel consumption is the same for both Hybrid and Gasoline; remaining mileage will differ for each type
- Create an abstract class named OnboardComputer with the two methods (first one concrete and second one abstract)
- Subclass OnboardComputer into OnboardComputerGasoline and OnboardComputerHybrid — notice abstract method has to be overridden
- Through composition the Car class will receive the OnboardComputer type during instantiation (i.e. it will be an input parameter in its constructor)
- Extend the Car class behavior by using the OnboardComputer type methods
- Create a test (client) procedure that instantiates a concrete onboard computer and a car and test the car's methods
- Bonus: extend the application by supporting Electric cars

Progress OpenEdge resources

- **Web site**

  https://www.progress.com/openedge

- **YouTube channel**

  https://www.youtube.com/channel/UCNxy1VY73tYJ7o_R25HjohQ/playlists?shelf_id=10&view=50&sort=dd

- **Documentation**

  http://documentation.progress.com/output/ua/OpenEdge_latest/

- **Technical support and knowledge base**

  https://www.progress.com/support/reference-guide

- **Developer community**

  https://community.progress.com

- **Progress eLearning community**

  https://wbt.progress.com