I spent an afternoon playing around with the code and made a lot more changes than I had planned to, I have included some notes on the different strategies that I used and also some data on the changes in the code quality and maintainability as a result.

**I managed to address points 1 and 3 (below) inside the first hour.**

Initial Thoughts  
  
Refactoring is an opinionated engineering exercise, you are unlikely to agree with all of my suggestions, but at the end of the day, they are only suggestions.

Normally, I would prefer to review the code first and do the refactoring as a pair-programming exercise. People generally produce smelly code for a reason and pairing creates lots of opportunities to talk through the code, understand some of the pressures, and to introduce changes in a positive way, rather than simply hacking at something somebody has worked hard on.  
  
I wouldn’t normally refactor a piece of code without having some examples of ‘good code’ to hand – e.g. this code uses a single underscore with an initial capital to denote a local variable and a double underscore for private variables, I can’t tell if this is your house style or not, so I just continued the pattern – I am trying to match how you would have done it.

I like to write naïve low-level code and throw a lot of exceptions – this forces the development team to validate and resolve data quality issues close to the source. If they don’t they will have to write try-catch blocks, which is a strong signal that there may be a mis-match between the UX and the model or that a data-source has become unreliable. There was no exception handling in this code, so I left it out, but it would be good to talk about how you handle errors.  
  
And finally, I am an experienced software engineer and this is a coding task; I tried not to get too fancy with the code but I use abstracts, generics and lambdas a lot in my work. If you don’t like them then it’s not a big deal, I can work within your guidelines.

The problems that stood out were:

1. PinacleClient and PartInvoiceController couldn’t be tested in isolation
2. Data access logic in the RepositoryDB classes
3. Overly complex methods

Over all it felt like the code could be consolidated a bit more – e.g. there was some logic in the PartInvoiceController to determine if a Customer existed, but “ID > 0 means a Customer exists” is an invariant fact about the Customer class and really ought to sit as an ‘Exists’ property within the class. I have tried to shift responsibilities between classes to keep logic as close to the data as I can.

Refactoring strategies

Introducing dependency injection (DI)  
  
Dependency Inversion is the last of the five SOLID principles and boils down to the idea that high-level components of a system (those closest to the conceptual model – business logic, workflows,…) shouldn’t depend on low-level components of a system (those closest to the physical model – data storage, network access,..). The way you achieve this by replacing concrete references with abstract ones (primarily through interfaces, although abstract base classes also work), dependency injection is a common design pattern that helps to achieve this through these practices:

1. Extract an interface for each low-level component.
2. Create a constructor in the high-level component that takes abstract references to the low-level components that it consumes.
3. Propagate this pattern down the chain – until there are no concrete references in the code.

I used DI in the PinacleClient and PartInvoiceController classes to inject an IDataService component. The default constructors inject concrete references to DataSerivce maintaining the *status quo*.  
  
DI has made the code a lot more testable, I have been able to build mock versions of each of the services to test the behaviour of the CreatePartInvoice method with out impacting any existing code.

### Composing dependencies

You can directly inject each component into a construct, but often the same set of components are collaborating together – composition is a good way to formalise these relationships. Passing a composite allows the mix of components to change without having to modify the consumers.

### Encapsulating the data access

I personally prefer to abstract away any data/service access – people working in CustomerRepositoryDB and PartInvoiceRepositoryDB would need to know a lot about the underlying datastore to write a new method, this raises several issues:

1. Changes to a stored procedure will have to be manually propagated through the code
2. Unchanged code will still compile, but fail due to changes in the datastore
3. The development team will need to understand the architecture of the datastore
4. The code is tightly coupled to the nature of the datastore
5. Allowing properties to be manually mapped complicates testing
6. It will be difficult to maintain a consistent approach to data access between teams

I have replaced the inline data access logic with a call to a query object that encapsulates all of behaviour. This was a fairly hefty intervention, but the advantages of this approach are:

1. You only have one point of contact with each stored procedure, if the name of a parameter changes then you only have one thing to update in the code
2. Breaking changes in the behaviour of the stored procedure will break the code
3. The development team only need to instantiate the query class and execute it
4. The query class remains tightly coupled to the datastore, but the rest of the code doesn’t care, so you could easily change it and nobody would have to know
5. You only have to test one instance of the data access logic
6. It only takes one line of code to find a Customer or add an Invoice

If you had a more complicated system, then I’d suggest you use an ORM; Dapper is really good with simple operations like these and is easy to set-up. If you weren’t using stored procedures as much, then a UnitOfWork/Repository pattern might also be worth considering.

Each Query class inherits from the abstract AGenericQuery class which makes use of methods in the DatabaseHelper static class. We can chat about the merits of having two classes to interact with the database, but I felt that come common methods would be more reusable if they were in the DatabaseHelper class and I wanted AGenericQuery to be solely responsible for executing queries.

### Sharing responsibilities

The ‘S’ in SOLID stands for ‘Single responsibility principle’ – just do one thing well. The original CreatePartInvoice method violated this principle by:

1. Searching for a Customer
2. Checking stock levels
3. Implementing validation logic
4. Instantiating an invoice before adding it through the repository

I moved most of the logic into and overloaded version of CreatePartInvoice that took a Customer instead of a name, the original method is now solely responsible for finding the customer. The code for validating a customer is hard to read and is likely to be used elsewhere, I pushed the invariant customer validation logic back on to the Customer class to make it more accessible,

There is code that requests the availability of a stock item and checks if it is > 0 before creating an invoice, I wrote an extension method for IPartAvailabilityService to encapsulate this behaviour. A stock level > 0 seems like a bug – so EnoughOfTheProductIsInStock takes an optional minimunQuantity.

The validation logic is likely to change, consolidating and moving it into its’ own method will allow you to monitor the stability of the model overtime. It is clear what the OrderIsValid method is responsible for. Passing a Customer into the method it clear that it is part of the validation process and allows implement something more sophisticated in the future without having to chance the CreatePartInvoice method.

Creating an invoice up-front didn’t seem to add much value to the process, so I added an overloaded method to the PartInvoiceRepositoryDB class that took over the responsibility for creating the invoice object.

I added some factory methods to CreatePartInvoiceResult to clarify the intention of the return value.

## Outcomes

### Improved testability

Without access to the database it was impossible to write any tests prior to refactoring. Refactoring allowed me to construct ‘mocked’ representations of the data services, control their behaviour and test aspects of the PinacleClient off-line. I have been able to write tests to explore the OrderIsValid method and confirm that the CRM. GetCustomerByName => Customer mapping was working.

Additional testing would require access to the database, but having isolated the business logic from the services it is debatable if further testing would do anything other than test third party code?

### Improved code quality

Running ‘Code Metrics’ over the solution prior to refactoring is a good way to see areas in the code that could be improved. The methods that accounted for over half of the executable code, scoring the lowest for maintainability and highest for coupling were obvious candidates.

Before refactoring

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type | Member | Maintainability Index | Cyclomatic Complexity | Class Coupling | Executable Lines |
| PartInvoiceController |  | 55 | 5 | 5 | 16 |
| PartInvoiceController | CreatePartInvoice | 55 | 5 | 5 | 16 |
| CustomerRepositoryDB |  | 58 | 2 | 12 | 11 |
| CustomerRepositoryDB | GetByName | 58 | 2 | 12 | 11 |
| PartInvoiceRepositoryDB |  | 59 | 1 | 10 | 11 |
| PartInvoiceRepositoryDB | Add | 59 | 1 | 10 | 11 |

After refactoring

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type | Member | Maintainability Index | Cyclomatic Complexity | Class Coupling | Executable Lines |
| PartInvoiceController |  | 88 | 9 | 8 | 7 |
| PartInvoiceController | CreatePartInvoice | 74 | 2 | 4 | 4 |
| CustomerRepositoryDB |  | 93 | 1 | 3 | 1 |
| CustomerRepositoryDB | GetByName | 93 | 1 | 2 | 1 |
| PartInvoiceRepositoryDB |  | 89 | 2 | 4 | 2 |
| PartInvoiceRepositoryDB | Add | 88 | 1 | 2 | 1 |

Dependency injection increased the complexity of the PartInvoiceController slightly, but there are fewer lines of code and all of the other methods score 85+ with low coupling.

The data access encapsulation pattern had a significant impact on the quality of CustomerRepositoryDB and CustomerRepositoryDB – each method is literally one line of code, and this hasn’t simply shifted the complexity elsewhere – the queries and the base class all score 85+ with low coupling and none of methods in DatabaseHelper dips below 70, the complexity has been refactored away.

### Sustainability

I am pleased with the design of the data access queries – I think they solve a lot of maintenance problems at once and presents a lightweight way to interact with the database, this pattern should be really easy for people to get on-board with.

I think the mocked services and the structure of the PinacleClient test provide a good template for parametric unit testing with XUnit.

Hopefully I have demonstrated the value of splitting complex methods into smaller ones sharing responsibilities between them.

SOLID can seem like an onerous set of principles, often expressed in a terse technical language but they are pretty easy to follow and they allow you to insulate people from some of the underlying complexities of a solution while encouraging you to build it in a way that makes it more testable and easier to work with.