Test Driven Development

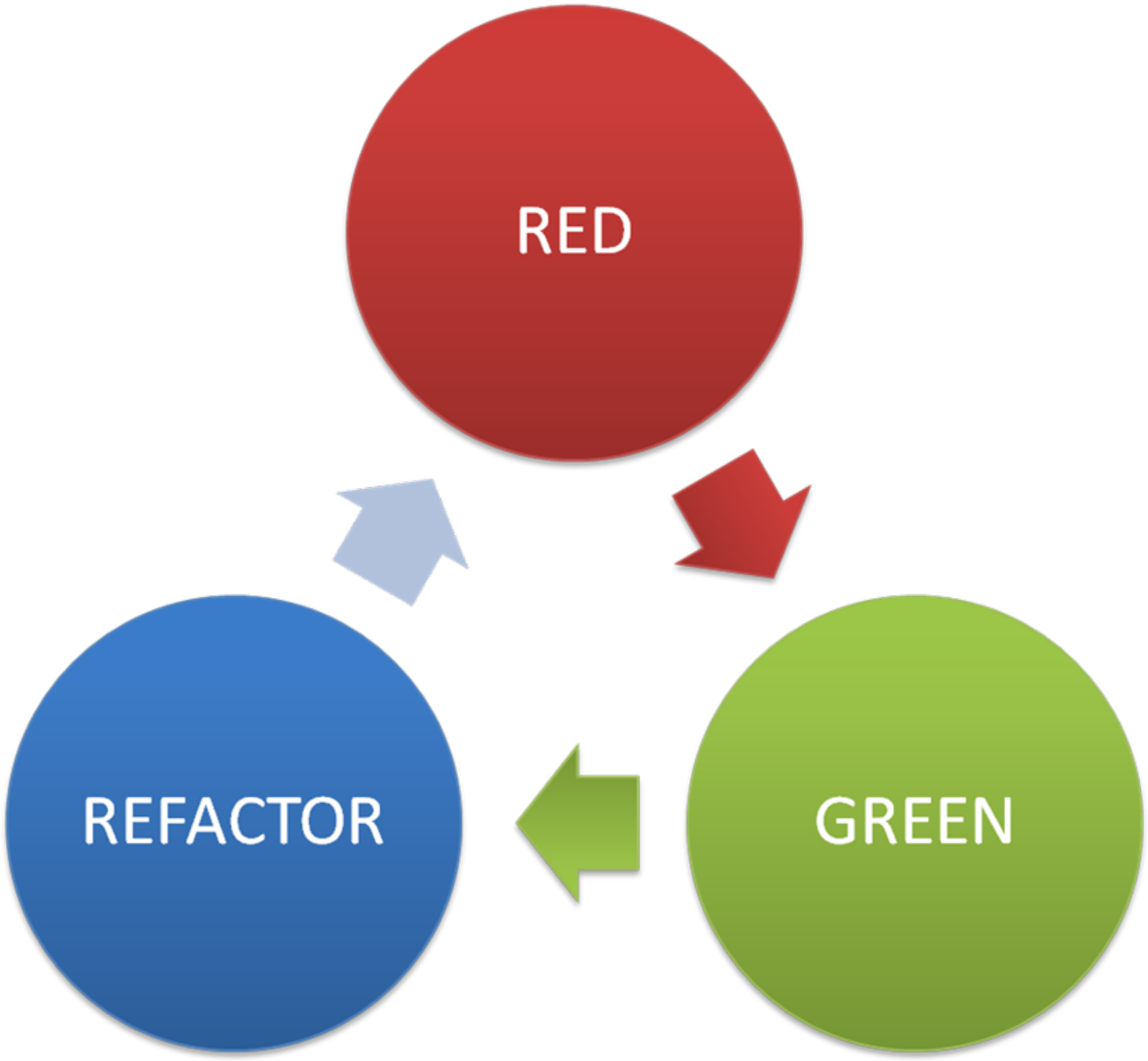
Demo och tips

Varför är det viktigt att vi skriver enhetstester i detta projekt?

- Vi är väldigt många inne i samma kodbas - enhetstester underlättar att förstå och ändra kod

- Snabbare utveckling - man får reda på om har haft sönder någon funktionalitet vid ändringar, samt att det man utvecklar fungerar enligt spec

- Man är inte beroende av en annan person som behöver förklara hur dennes kod är tänkt att fungera


RED
GREEN
REFACTOR

Fördelar med TDD

- Man behöver tänka igenom ordentligt hur man ska skriva sin kod innan man börjar, vilket leder till tydligare och elegantare kod
- Skriver man tester före så blir det per automatik 100% testtäckning
- Ingen död eller överflödig kod - enbart kod som behövs för att uppfylla givna kriterier kommer att skrivas
- Koden blir mer modulär - annars går den inte att testa
- Många buggar elimineras tidigt utan behov av manuella tester
- Snabbare utveckling - man vet när man är klar
- Lättare att underhålla, refaktorera och lägga till ny kod tack vare ökad trygghet hos utvecklaren
- Bevis på att ens kod faktiskt implementerar den funktionalitet som är tänkt, ibland långt innan den används i applikationen
- Mindre komplex kod - behöver man skriva många testfall med många kriterier är det ett tecken på att koden är för komplex och gör för mycket och bör då brytas ut i fler klasser. Därmed ger TDD en bra indikation om koden är bra designad och enkel att förstå
- Vid buggrättning - skriv test innan som återupprepar buggen så kommer buggen aldrig att dyka upp igen!

Nackdelar

- Kan till en början hämma utvecklingstakten innan man har kommit in i det, vilket kan kännas frustrerande

- Alla måste uppdatera och lägga till testfall vid kodändringar så att dom inte blir inaktuella eller så att kod inte längre har testtäckning

- Dåligt skrivna tester ger 100% testtäckning men kanske inte testar koden ordentligt - kan ge falsk trygghet

- Kan vara svårt att testa alla möjliga cases - buggar kan slinka igenom

- Testar bara så att din specifika enhet funkar enligt spec - men integrationer mellan klasser kan missas och kanske upptäckts först vid manuella tester. Dock är syftet med enhetstester just att testa en separat enhet

Bad practices

- State (fields och properties) i/på statiska klasser går inte att mocka. Det finns även en stor risk att tester som använder samma statiska klass sätter om state vilket kan få tester att faila obefogat. Util-metoder (rena funktioner) är dock ok att ha i statiska klasser

- Flera asserts i en testmetod - failar första asserten så körs inte de efterföljande

- Hårda kopplingar (klass istället för interface) mellan beroenden - syftet är att testa just din enhet, inte klasser runtomkring. Externa beroenden och states ska mockas bort och användas för att testa kriterier i dina testfall. Undantag är Util-klasser som räknas som implementationsdetaljer

- Testa implementationsdetaljer - testa bara det publika gränssnittet och förväntade svar/beteenden

- Specifika testförutsättningar givna i en basklass (GivetEn*). I en testbasklass ska bara mockningar och uppsättning ske (dummy-data bör genereras från fall till fall)

Demo

Innan vi börjar

- Inget GUI finns - den enda valideringen av funktionalitet är enhetstester
- Arkitekturen efterliknar hur vi jobbar idag, men andra verksamhetstermer används
- Koden är implementationsfattig - enbart vad som behövs för denna presentation är implementerat
- Exemplet är onödigt komplicerat för att visa den stora nyttan med att skriva enhetstester
- Fantasi behövs :)

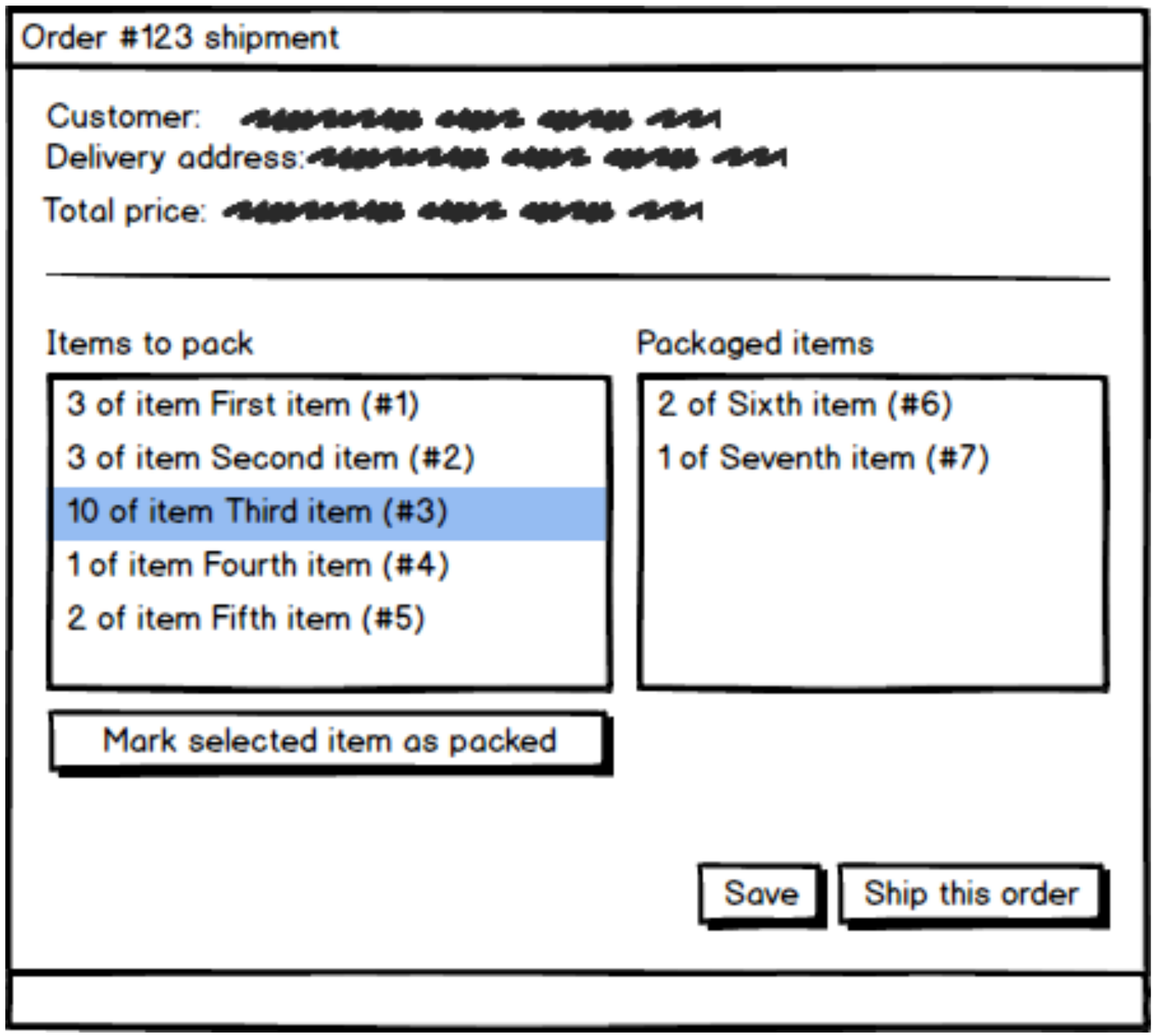
Agenda

- Vad det är för slags applikation
- Hur applikationen ser ut idag
- Hur applikationen ska se ut efter kravändringar
- Hur domänen som ska användas är uppbyggd
- Hur ändringar förs in genom att använda TDD(-ish)

Vad är det för applikation?

- Simulering av en webshop
- En kund gör en order (väljer flera artiklar)
- Vi paketerar ordern
- Vi skickar ordern
- I den här presentationen koncentrerar vi oss på den vy som ska hjälpa till vid paketering

Före kravändringar



Nytt krav

- Kunden ska kunna ändra ordern innan den är skickad, vilket går idag

- Ändringarna i ordern ska visas i GUI, vilket inte görs idag

- Ändringarna ska visas i den prioriteringsordning som visas i mockup-bilden (efter ändringstyp och sen i bokstavsordning)

Efter kravändringar

Order #123 shipment

Customer: [redacted]
Delivery address: [redacted]
Total price: [redacted]

The following items in the order has changed:
The quantity of First item (#1) has changed from 3 to 2
The quantity of Third item (#3) has changed from 10 to 8
A new order of 2 Nineth item (#9) has been added
A new order of 8 Eigth item (#8) has been added
The fifth item (#5) should no longer be packaged
The fourth item (#4) should no longer be packaged
The sixth item (#6) has been packaged but is no longer in the order

Items to pack

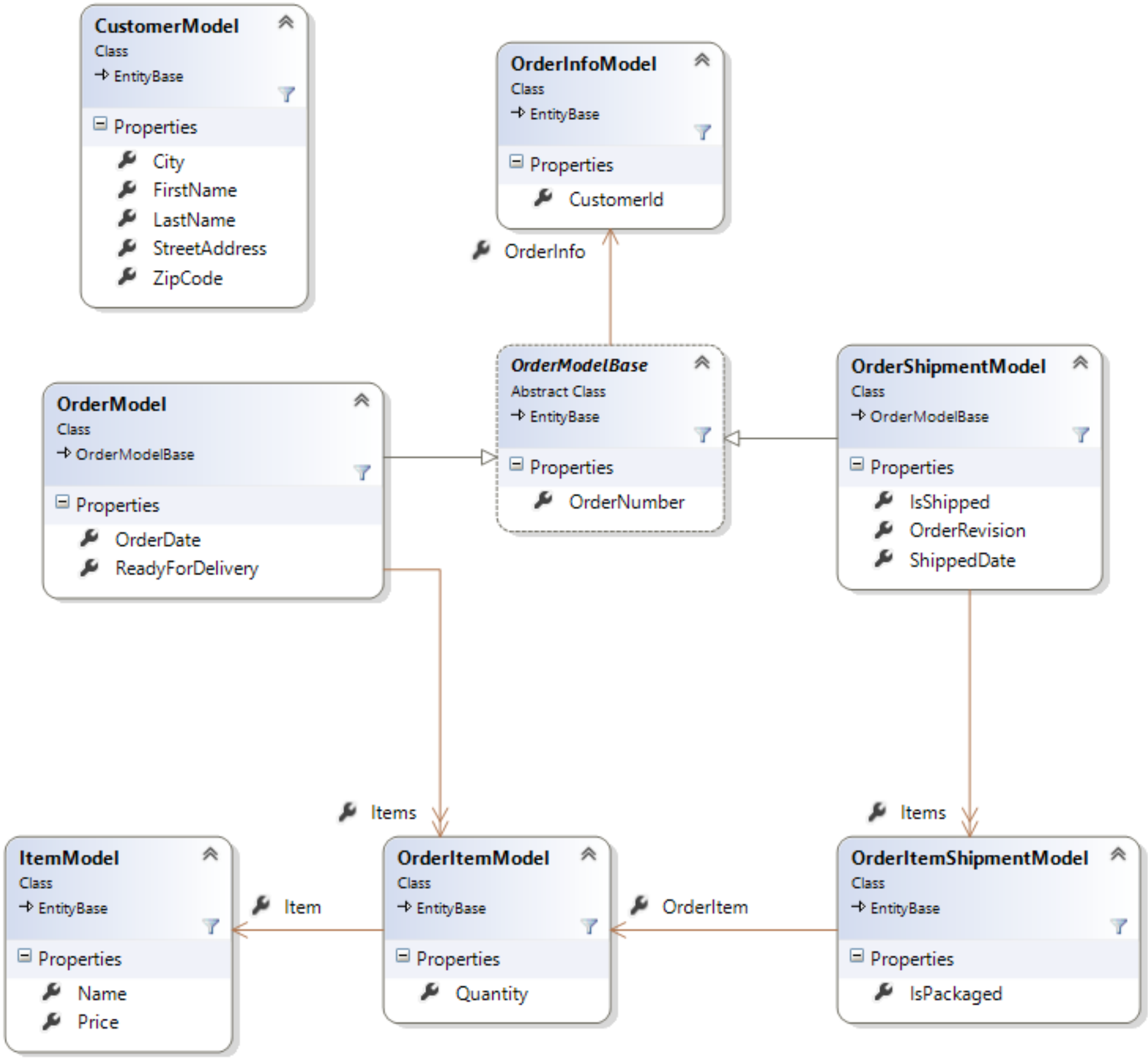
- 2 of item First item (#1)
- 3 of item Second item (#2)
- 8 of item Third item (#3)
- 8 of item Eigth item (#8)
- 2 of item Nineth item (#9)

Mark selected item as packed

Packaged items

- 1 of Seventh item (#7)

Save | Ship this order


CustomerModel
Class
EntityBase
Properties
City
FirstName
LastName
StreetAddress
ZipCode
OrderInfoModel
Class
EntityBase
Properties
CustomerId
OrderInfo
OrderModelBase
Abstract Class
EntityBase
Properties
OrderNumber
OrderShipmentModel
Class
OrderModelBase
Properties
IsShipped
OrderRevision
ShippedDate
OrderModel
Class
OrderModelBase
Properties
OrderDate
ReadyForDelivery
Items
Items
ItemModel
Class
EntityBase
Properties
Name
Price
Item
OrderItemModel
Class
EntityBase
Properties
Quantity
OrderItem
OrderItemShipmentModel
Class
EntityBase
Properties
IsPackaged

Order #123 shipment

Customer: [redacted]
Delivery address: [redacted]
Total price: [redacted]

Items to pack

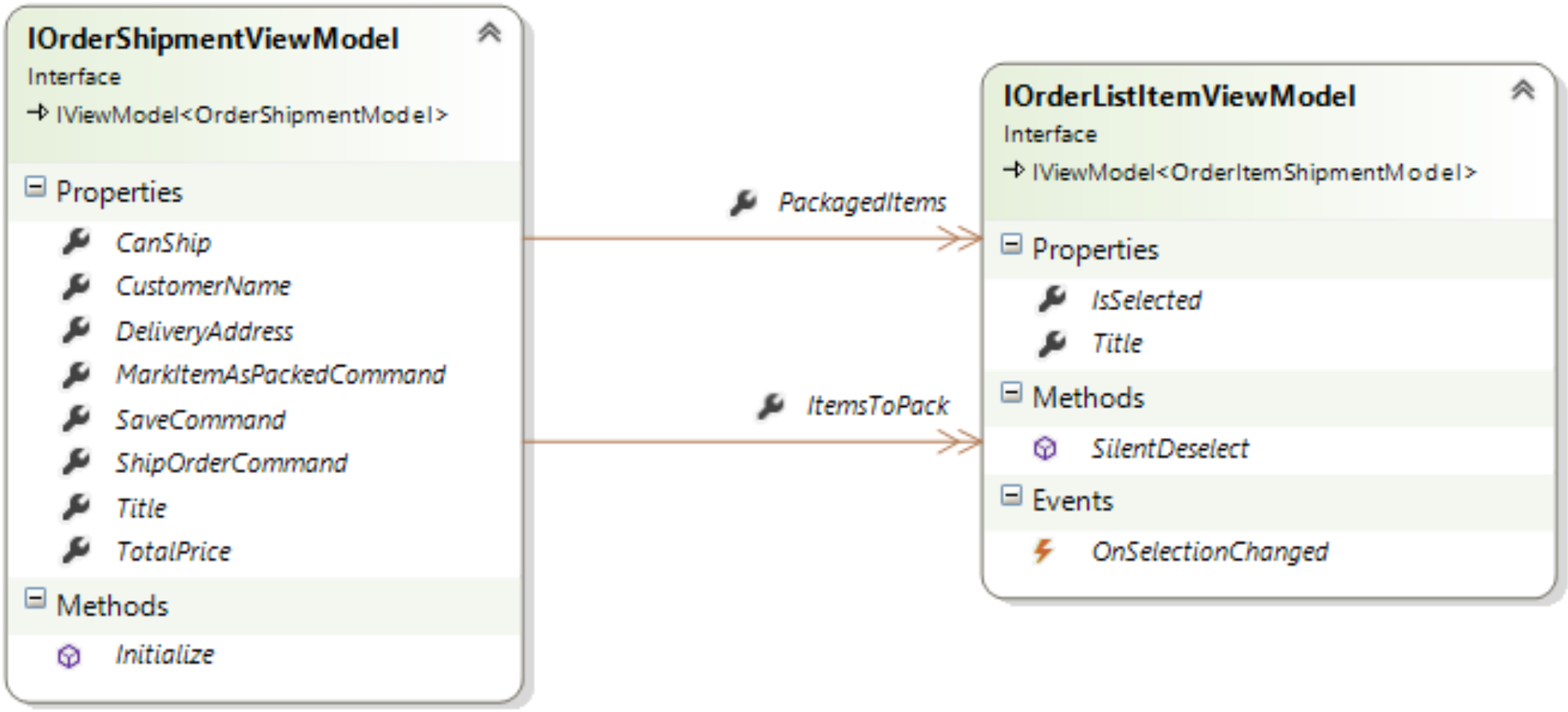
- 3 of item First item (#1)
- 3 of item Second item (#2)
- 10 of item Third item (#3)
- 1 of item Fourth item (#4)
- 2 of item Fifth item (#5)

Packaged items

- 2 of Sixth item (#6)
- 1 of Seventh item (#7)

Mark selected item as packed

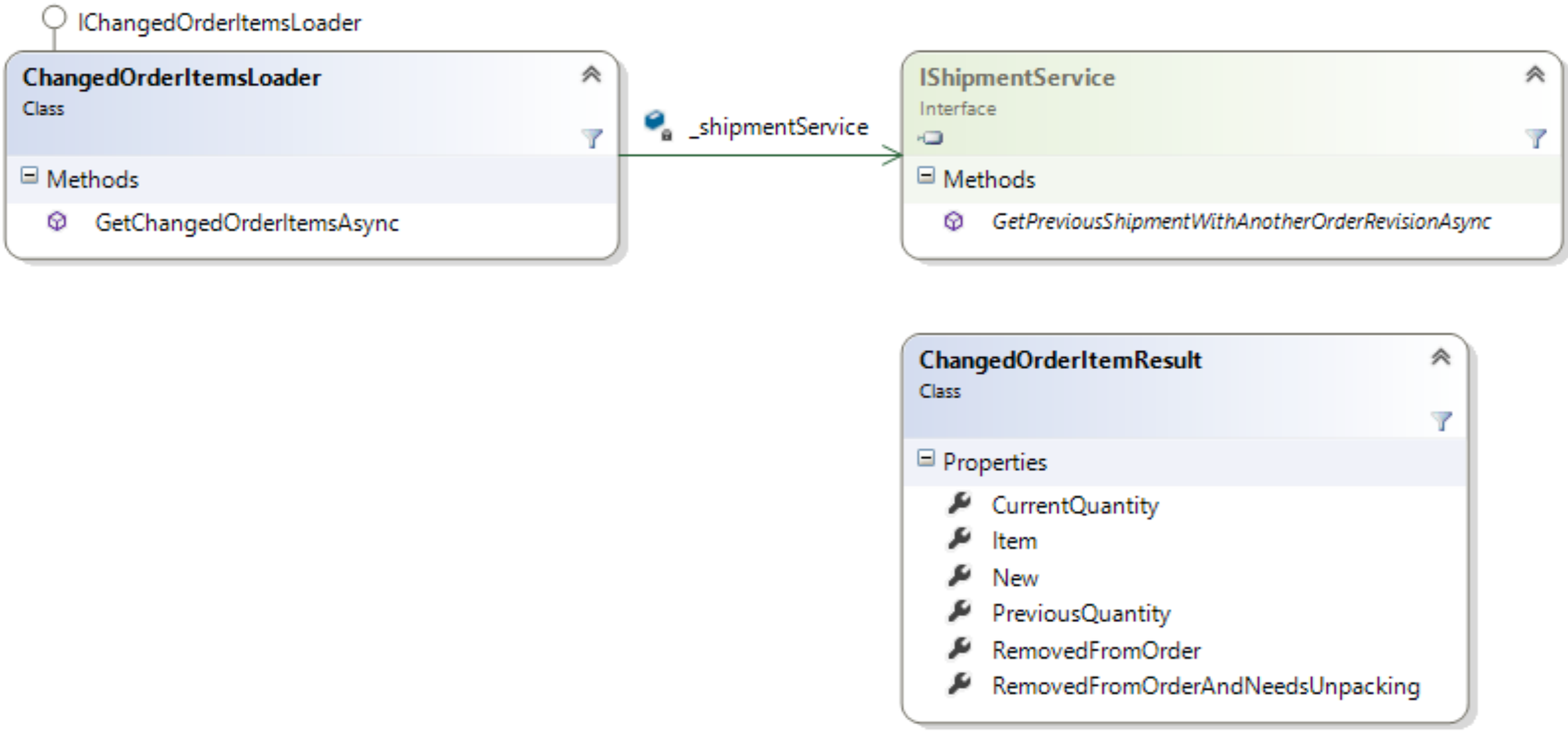
Save | Ship this order



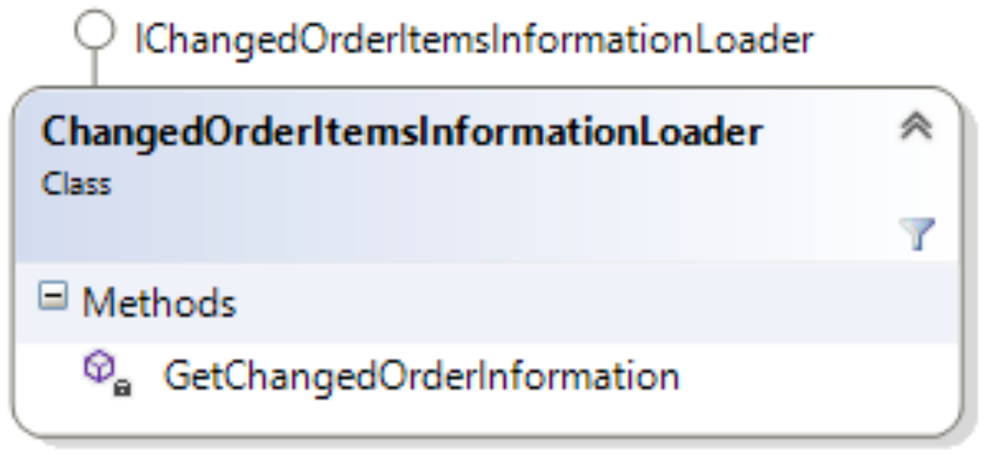
Vad som behöver göras

- Hämta ut vad som är ändrat mellan två revisioner av en order
- Transformera ändringarna till en informationstext som kan användas i vymodellen
- Anpassa vår klass som hämtar data så att den hämtar ut denna information för vymodellen
- Anpassa vymodellen så att den använder informationen

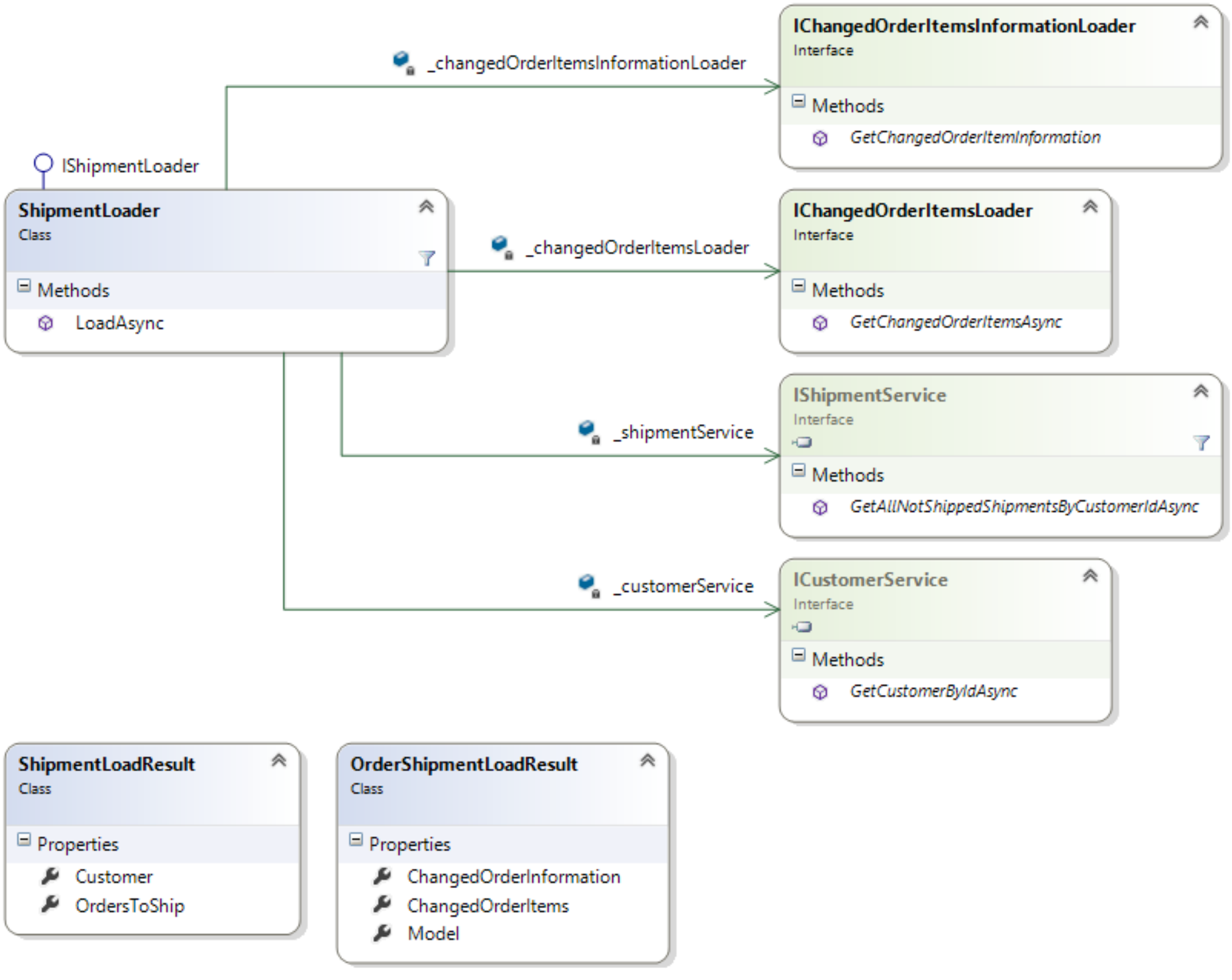
Hämta ut vad som ändrat mellan två revisioner av en order



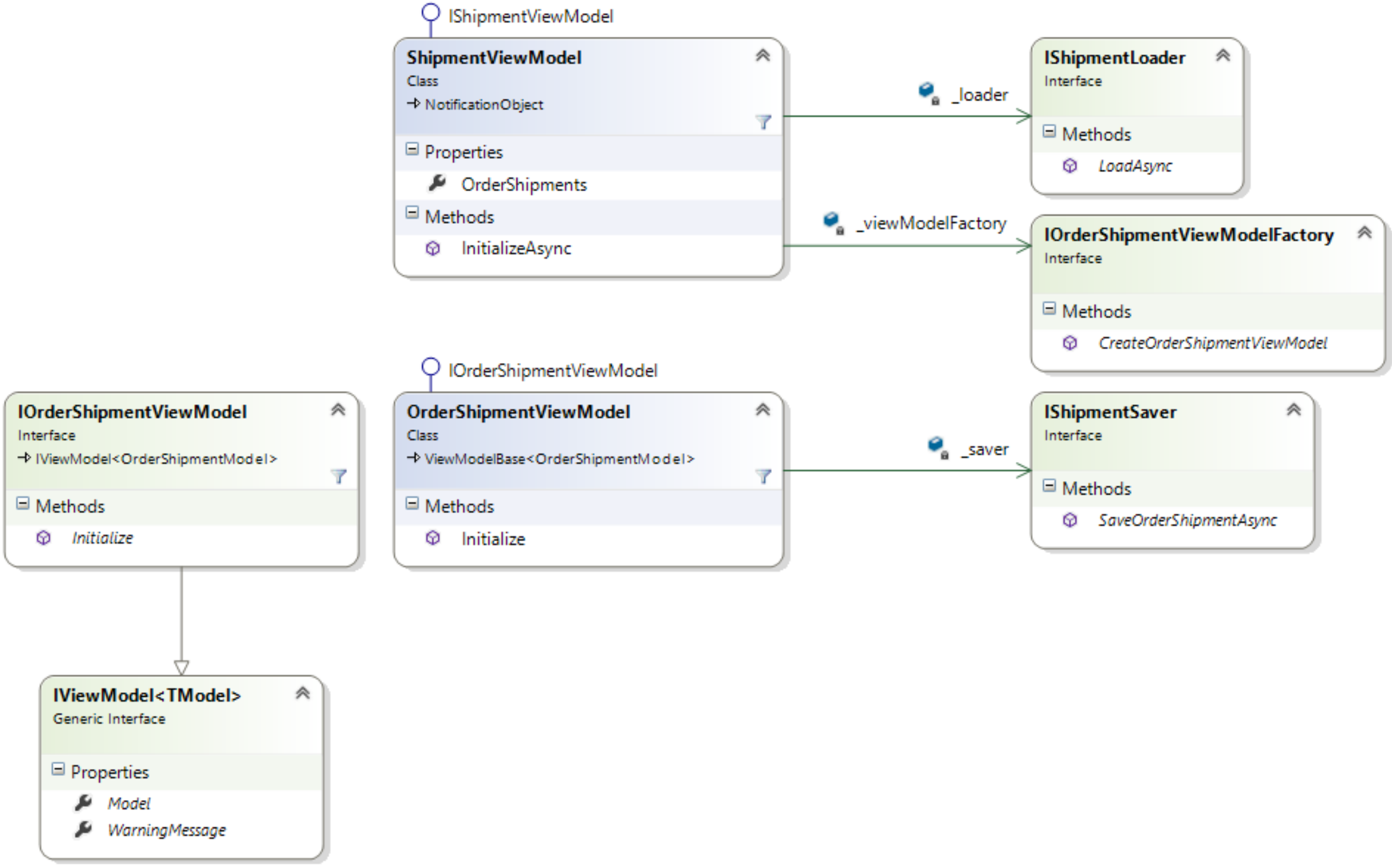
Transformera ändringarna till en informationstext som kan användas i vymodellen



Anpassa vår klass som hämtar ut data så att den hämtar ut informationen för vymodellen



Anpassa vymodellen så att den använder informationen



Tack för visat intresse

http://www.github.com/bjorse/tdd-demo