7SENG001 Enterprise Application Development

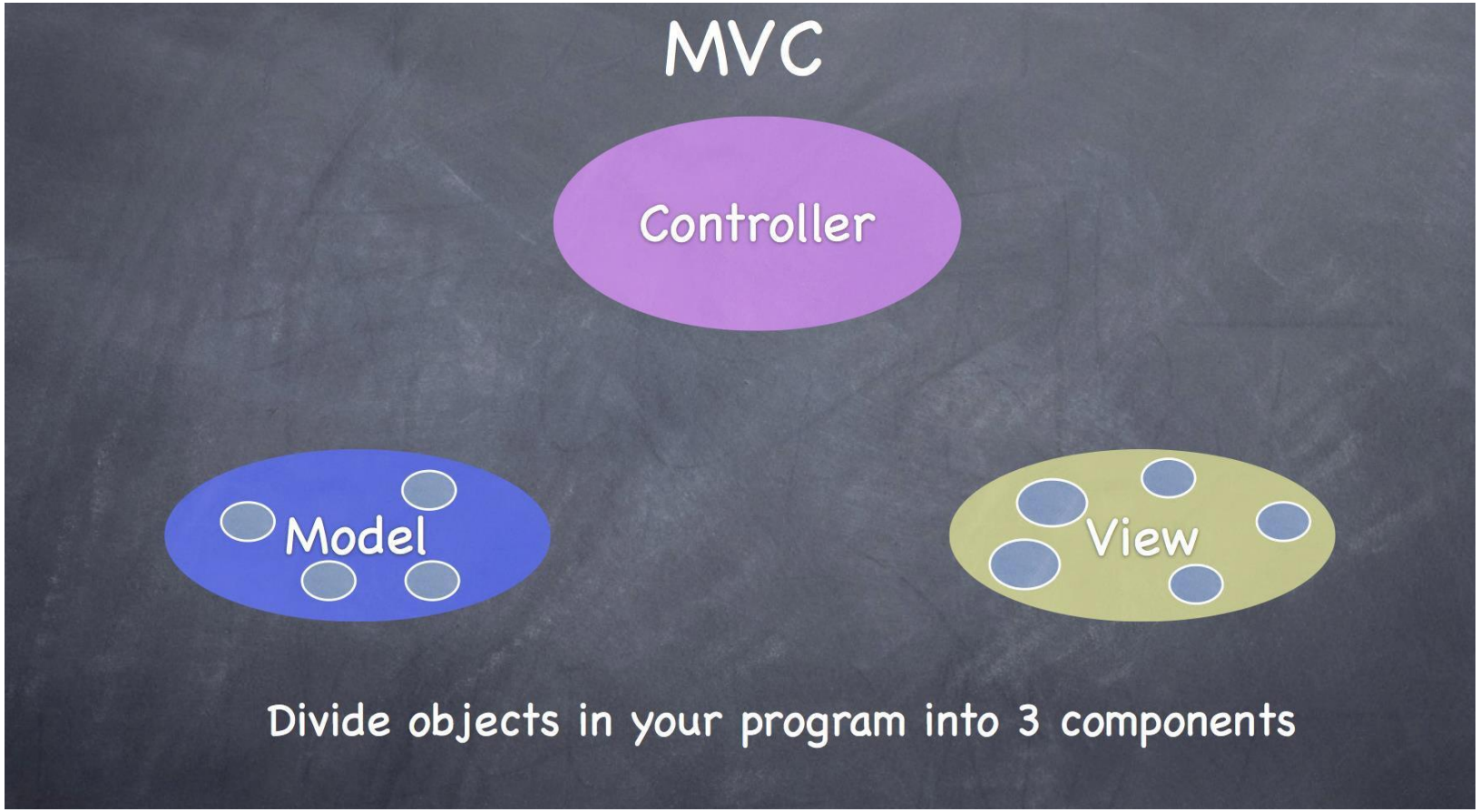
Modelling , UML In Depth

Week 3

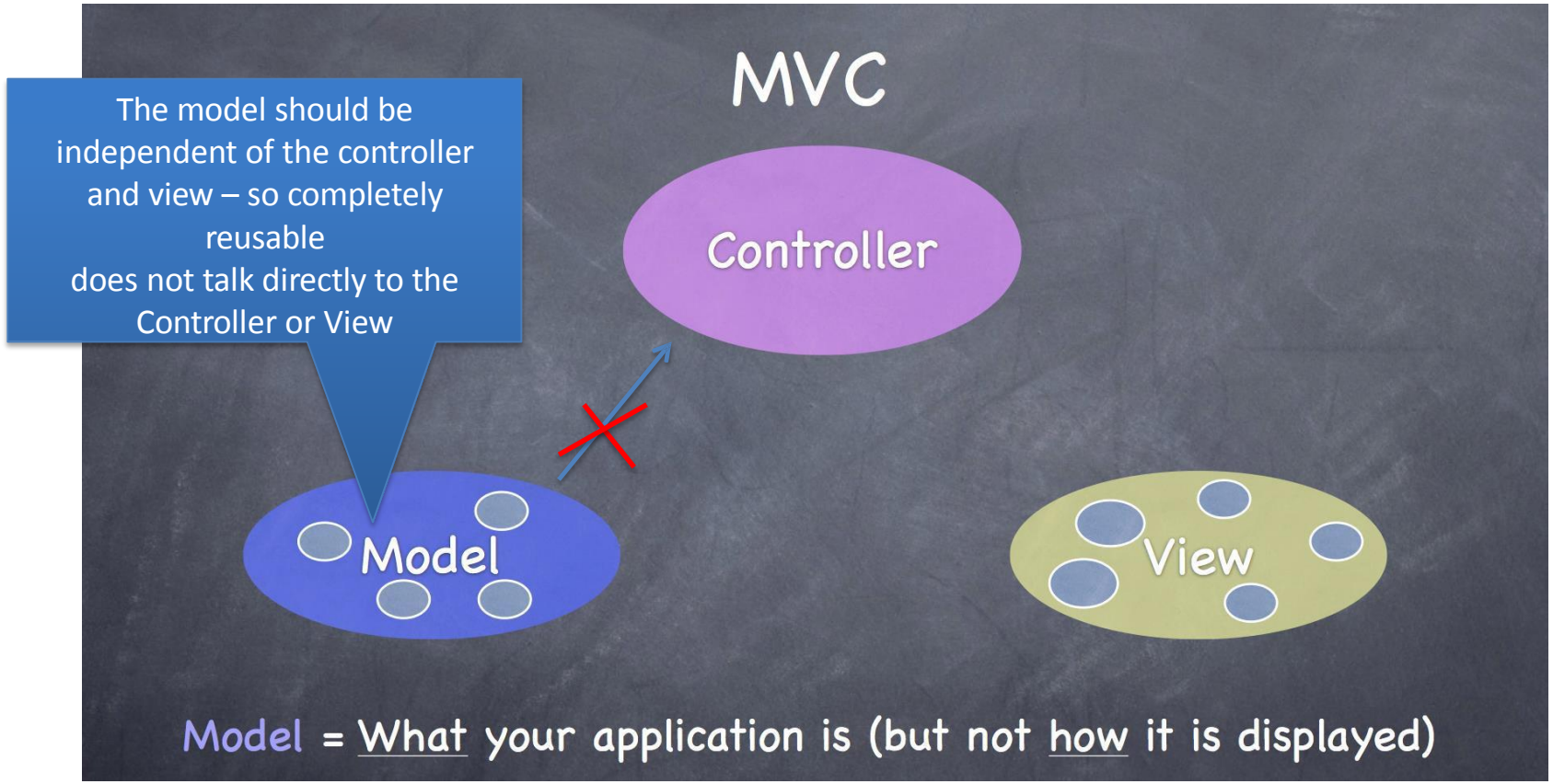
Object Oriented Design Patterns

The Model View Controller

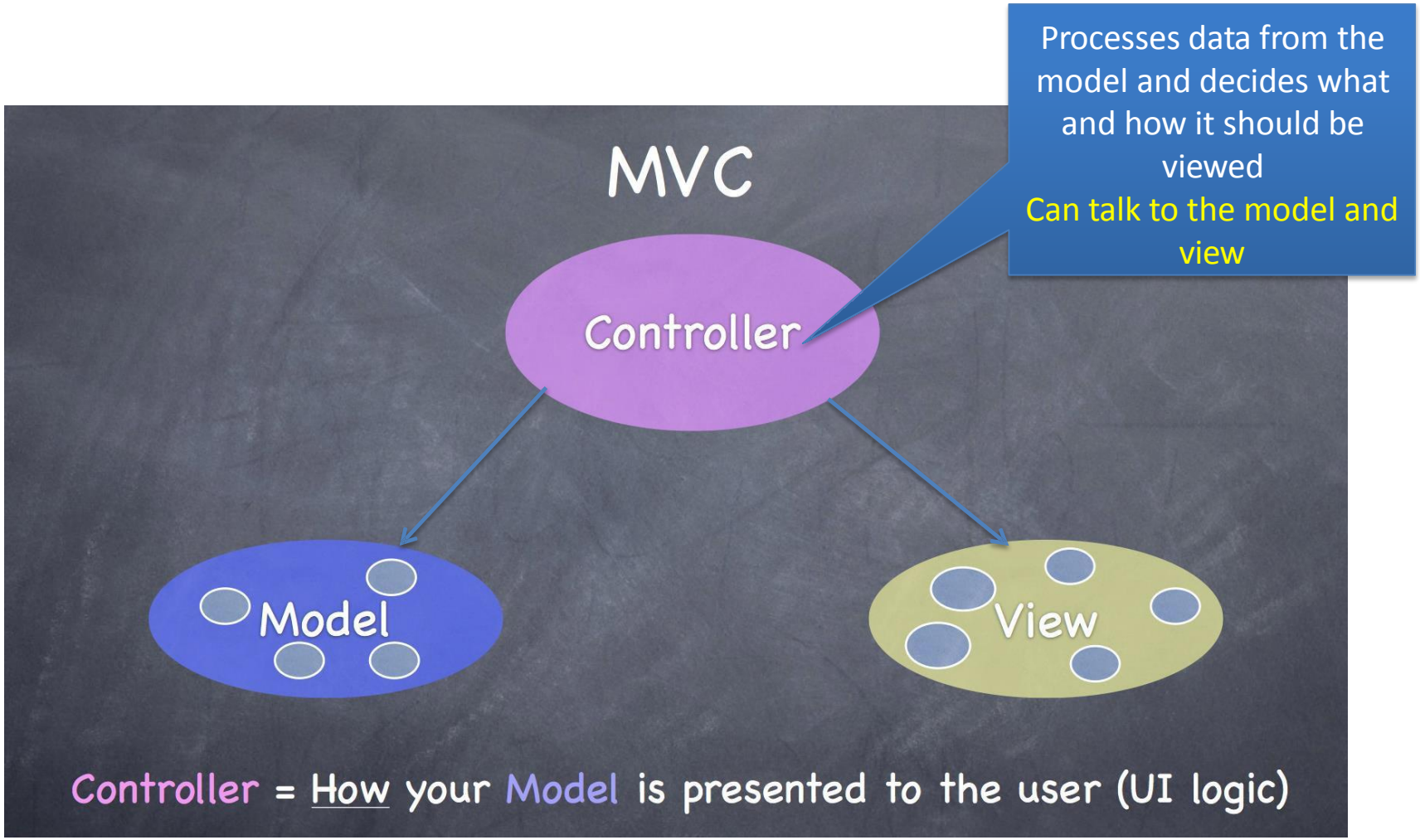
(Extensible pluggable framework)



Model


MVC
The model should be independent of the controller and view – so completely reusable
does not talk directly to the Controller or View
Controller
Model
View
Model = What your application is (but not how it is displayed)

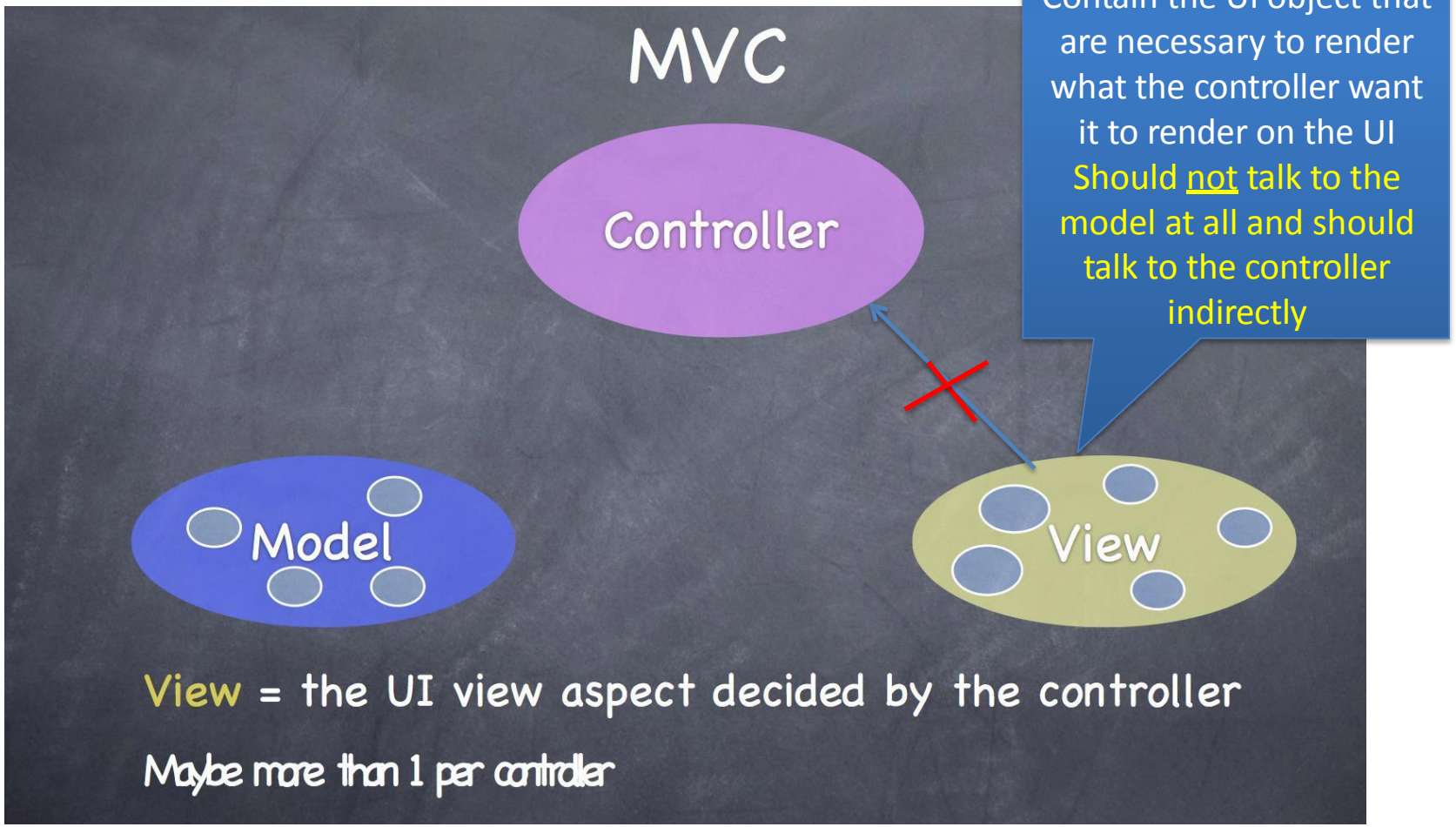
Controller



Processes data from the model and decides what and how it should be viewed

Can talk to the model and view

View

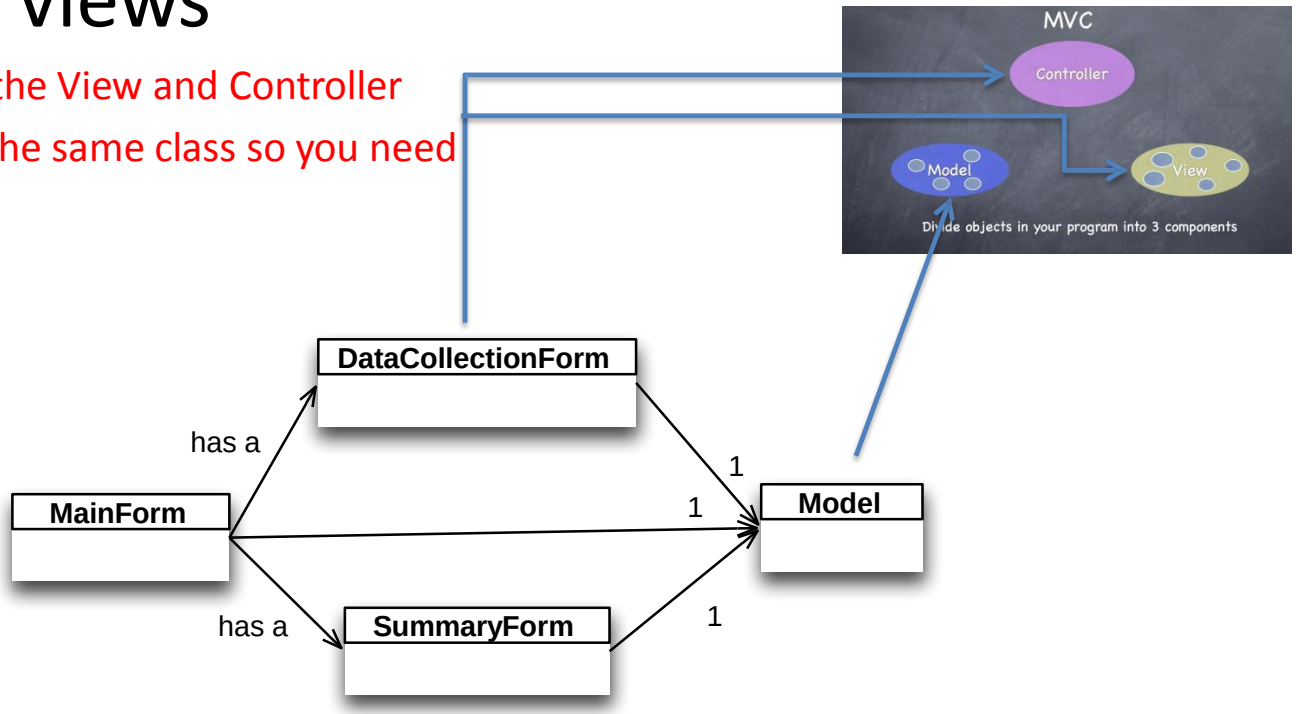

MVC
Controller
Model
View
Contain the UI object that are necessary to render what the controller want it to render on the UI
Should not talk to the model at all and should talk to the controller indirectly
View = the UI view aspect decided by the controller
Maybe more than 1 per controller

Example

MVC Pattern – The Model

Model should be independent and not know about the views

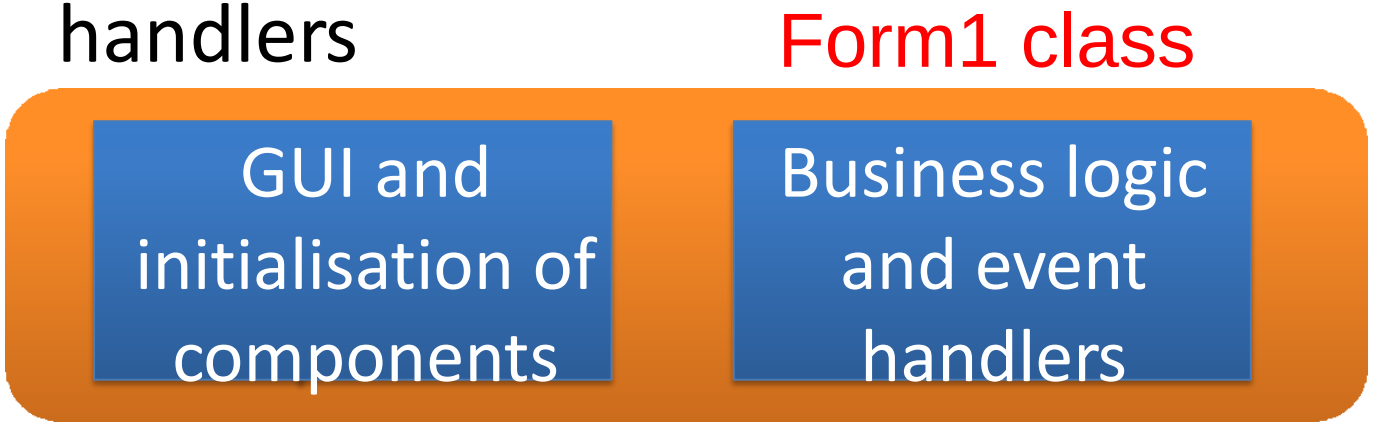
In Windows Forms the View and Controller generally reside in the same class so you need to be aware of this



Remember the Partial Class

- This is used to logically break down the class into:

1. Code associated with GUI and application set-up
2. Code that implements behaviour and event handlers



The Partial Class reminder

- A class in C# can be split over two files by using the partial class keyword

```
namespace WindowsFormsPartialClassExample
{
    public partial class Form1 : Form
    {
        public Form1()
        {
        }     //this method is in the other the class
    }         InitializeComponent();
}
```

This is in Form1.cs

Partial Class reminder

```
namespace WindowsFormsPartialClassExample
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer
  components = null;

        /// <summary>
        /// Clean up any resources being used.

...
```

This is in Form1.Designer.cs

Domain Modelling

- 1. How does a designer arrive at the set of classes that make up the fundamental architecture of a system?
- 2. How do you know that a given set of classes will actually support the required functionality?
- Questions like this are answered by the “process” component of a methodology

Two Approaches

- There are two main approaches that have been adopted to the problem of getting started with the design of a new system
- **Data-driven approaches**
  - These approaches look first at the data that is to be handled by the system, try to model it as a set of classes, and then implement the required functionality on these classes
- **Behaviour-oriented (functional) approaches**
  - These approaches look first at the behaviour, or functionality, required of the system, and try to deduce from this what classes are required to support this behaviour

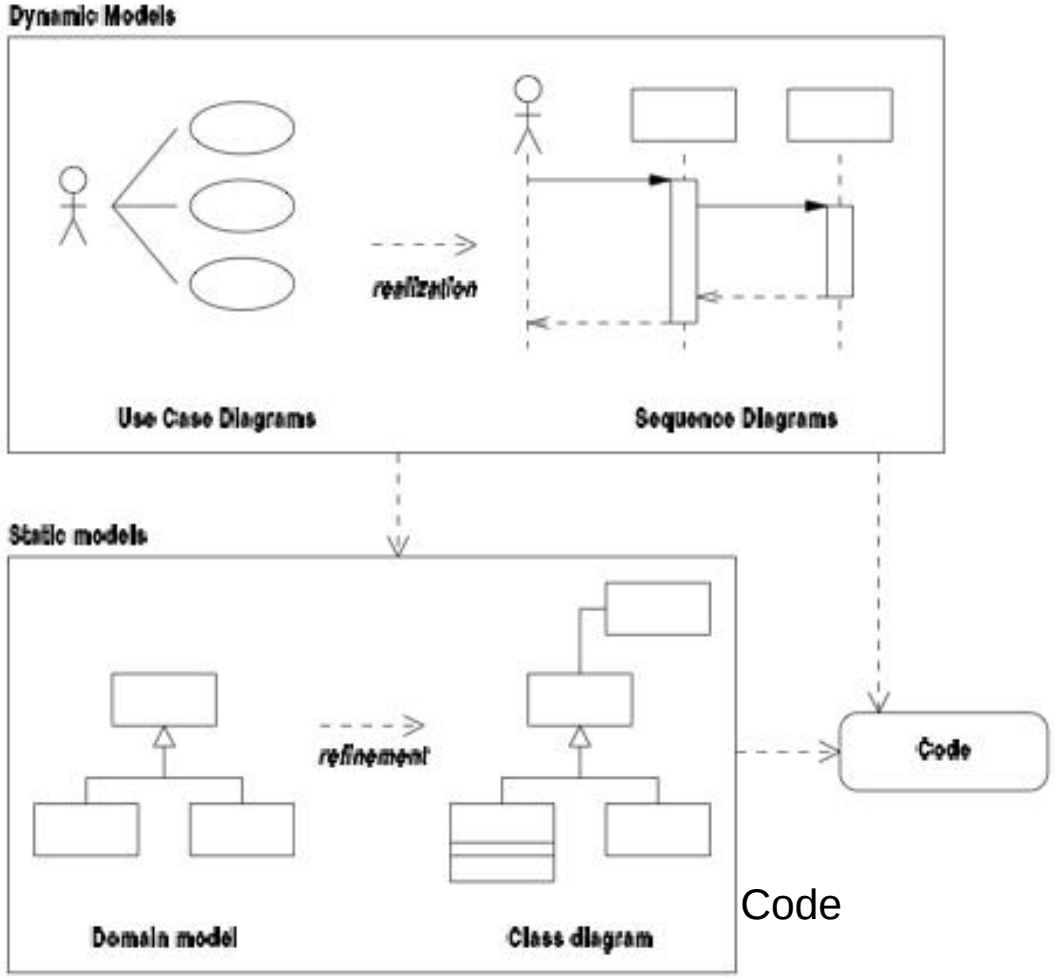
Other Approaches - View

- Some applications are strongly linked to forms it is likely that some aspect of the analysis should be therefore linked to views
  - Visual Basic/C# programmers often put all of the model and business logic in forms
- Views should be driven by use cases
- It is always advisable to mock-up and paper-prototype views
- Follow the **Model View Controller (MVC)** pattern
  - Sometimes control and view are mixed in the same class

The process

Dynamic Models

Static Models



Code

Static Models

- Static models describe the structure of the system
- **Domain models** describe the problem space: the “real-world” entities and data that the system will deal with described in terms of the object model
- The structure of the solution space (the program) will be different and usually more complex, but hopefully the basic structure of the domain model will still be visible in the final code.

Dynamic Models

- These describe the system’s behaviour:
- Use case models classify the users of the system and the various activities that they can perform with it
- **Sequence** diagrams break these high-level actions down into detailed interaction between objects
- State-charts summarize dynamic properties of objects in a given class

Design Consistency

- The structure specified in the static models must support the interactions specified in the dynamic models
- For example:
  - the objects shown on **sequence diagrams** must be **instances of classes given in the class model** that enable the interaction
  - Note this is not necessarily true for analysis diagrams

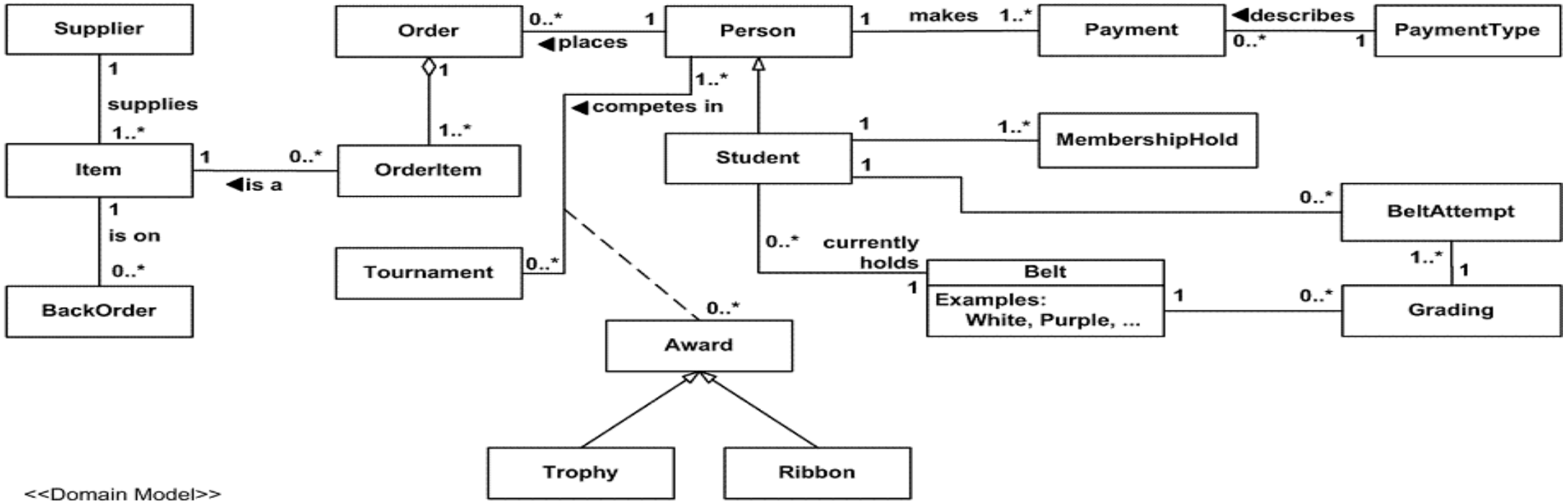
Use Case Driven

- UML is often described as supporting use-case driven methodologies
  - This means that development starts with the production of a use case model, to capture the system requirements, and that subsequent work depends heavily on the information in this model
- This is intended as a way of ensuring that the final system does in fact satisfy the users' requirements

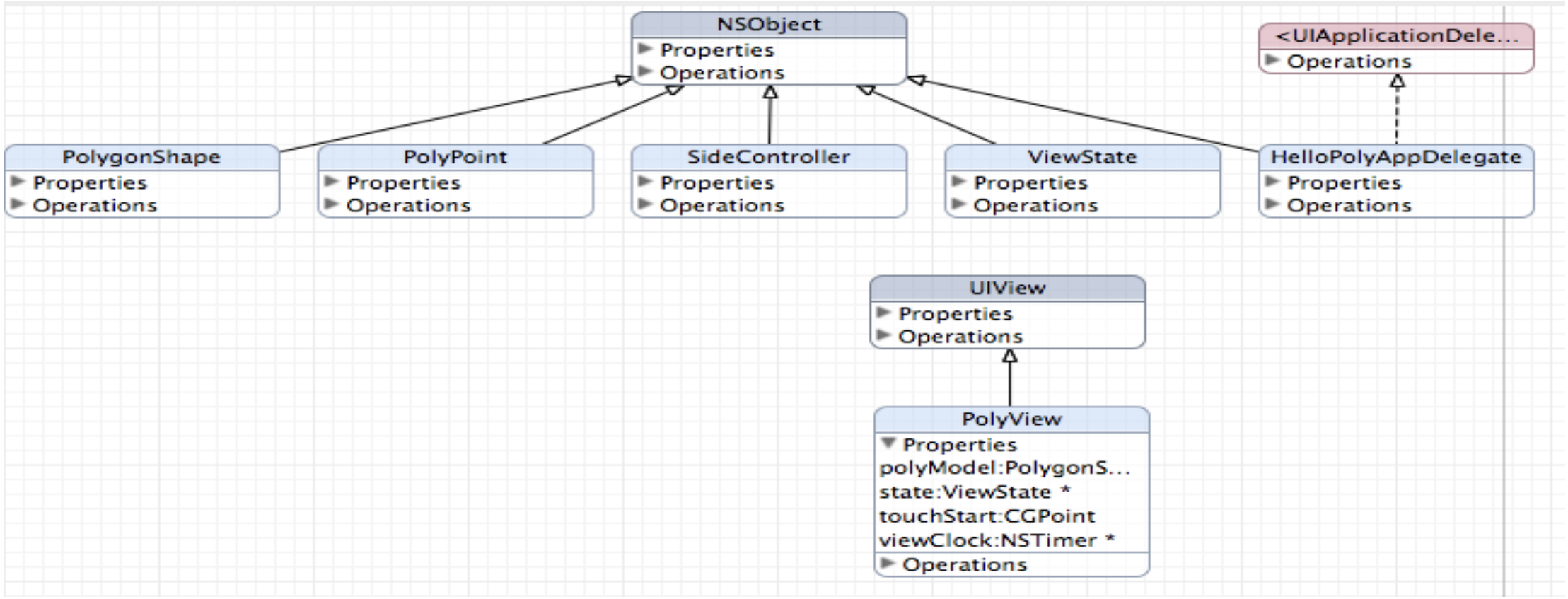
Domain Modelling

- A domain model is an **initial class diagram** which is intended to capture preliminary thoughts about the classes involved in an application, and the relationships between them – these are often real-world entities
- The intuitions guiding this process are:
  - 1. Classes model real-world entities (an entity is anything that we need to store data about, and could be concrete or abstract)
  - 2. Associations are used to model the relationships between these entities

A Domain Model Example



A reverse engineered class diagram



Differences

- It is clear that the domain looks quite different to a design class diagram
- The class diagram in the previous slide shows the actual classes developed for the application and is therefore very useful for **understanding and maintaining** the software
- Whereas the domain model is very useful for helping **analyse** the domain problem and eventually **develop** the software

Domain Modelling

- A domain model provides a starting point for subsequent work, and in most cases will be significantly revised as the design evolves. Ultimately, it will evolve into a complete class diagram that can be used as a basis for implementation

Finding Classes

- Classes in a domain model often correspond to “things” that are mentioned in a description of a system
  - It can be helpful to **pick out nouns** from a written specification as a way of constructing an initial list of classes
- Classes partition the data space of a system, so it is often useful next to identify all the separate data items that must be recorded by the system, and to allocate each as an attribute of some appropriate class
  - Doing this will often cause revisions to the original list of classes.

Finding Classes

- When looking for classes, watch out for
  - **Synonyms**
    - phrases that mean the same thing, in the context of the system
  - **Irrelevance**
    - things that are outside the scope of the system
  - **User-interface concepts**
    - the details of the system's user interface are not part of its domain, and should be ignored at this stage

Finding Relationships

- There are two major kinds of relationship between classes that appear in domain models
  - **Generalization**
    - If one entity can be viewed as a special kind or type of another, consider linking them with a generalization relationship. If two classes share many attributes, consider creating an abstract superclass to bring out this commonality
  - **Associations**
    - All other relationships between entities should be modelled as associations. Take care to label each association in an informative manner, and consider multiplicities where there is sufficient information to do so

Ignore behaviour

- It is very hard to identify sensible operations for classes from an examination of a written specification
- Operations are identified as a result of **use case modelling**, and so domain models normally do not show any operations on classes

Domain Modelling

- One commonly suggested technique for getting an initial set of classes is by performing a simple linguistic analysis on the specification, or other available material
- List the following:
  - **Nouns**
    - These often represent classes or attributes
  - **Adjectives**
    - These often represent attributes of the nouns they qualify
  - **Verbs**
    - Sometimes represent associations rather than operations.

Refining

- Initial lists of nouns, adjectives and verbs should not be taken at face value
- Different words or phrases can sometimes describe the same thing. These should be identified, and a single term chosen
- Things mentioned in a specification may be irrelevant to the actual system being developed, or part of its environment rather than the system itself. These can simply be ignored
- Verbs should be carefully classified into those which describe relationships and those which describe actions. Only the **relationships** are required for a **domain analysis**

Domain Model Conclusion

- Drawing an informal domain model can be a useful way of getting into the design of a new system
- However, the production of such a model provides no guarantee that it will adequately support the functionality of the system
- Producing dynamic models, and in particular sequence diagrams, will enable checking and refinement of the initial domain model

In-class exercise

Create a **domain model** for a library from the following librarian description.

“Our library has a catalogue which lists a variety of media including Books, DVDs, and CD’s. Items are loaned to our users, each user has the ability to loan up to 12 items”

- Underline the Nouns and verbs. Identify Classes and associations.

Simple Library Domain Model