ECE4574 – Large-Scale SW Development for Engineering Systems
Lecture 3 – Object-Oriented Design

Creed Jones, PhD

Course Updates

- No class next Monday (Labor Day)
- Homework 1 is due Friday, September 22
- Quiz 1 is This Wednesday, 7 PM to midnight Eastern time
  - Look in the Quizzes section of Canvas

Graduate Teaching Assistant

Varun Modak

Office hours, in-person and via Zoom:

- Monday: 5:30 - 7:30 PM (Online)
- Tuesday: 10:00 AM - 1:00 PM (In-person)
- Wednesday: 1:30 - 3:30 PM (Online)

Online hours are at zoom ID# TBD

In-person hours are in TBD

Topics for Today

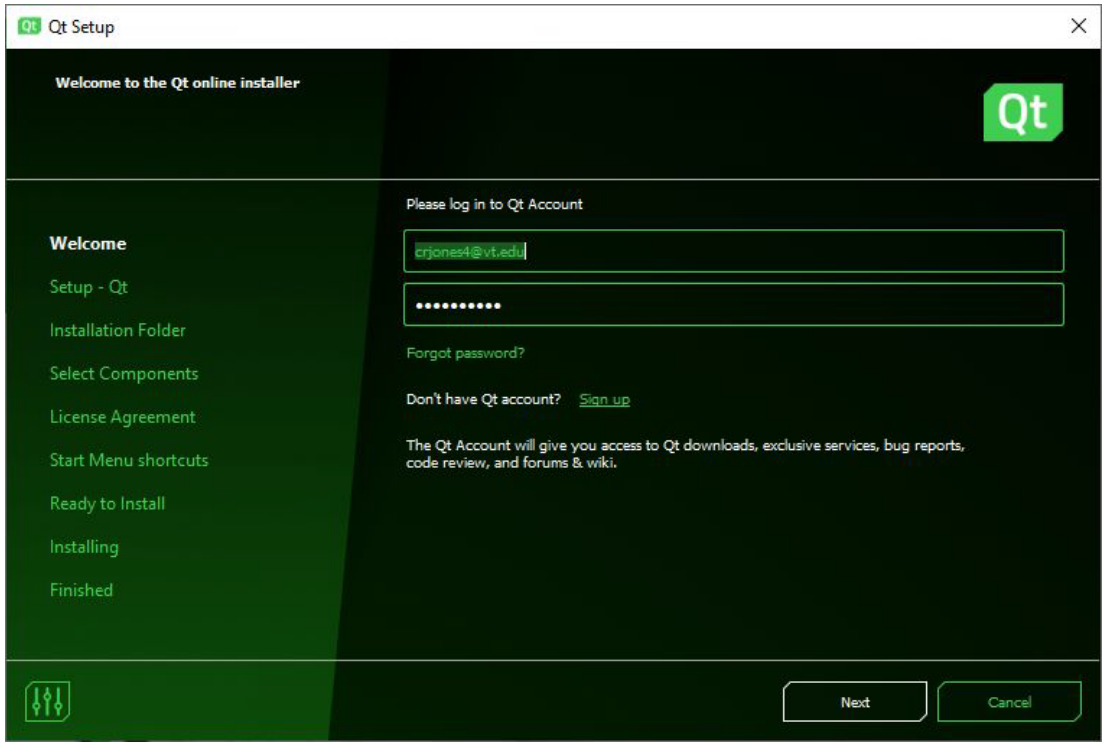
Installing Qt

Unified Modeling Language

- UML Concept
- UML Diagrams
  - class
  - interaction
  - component
  - use case
- Simple example

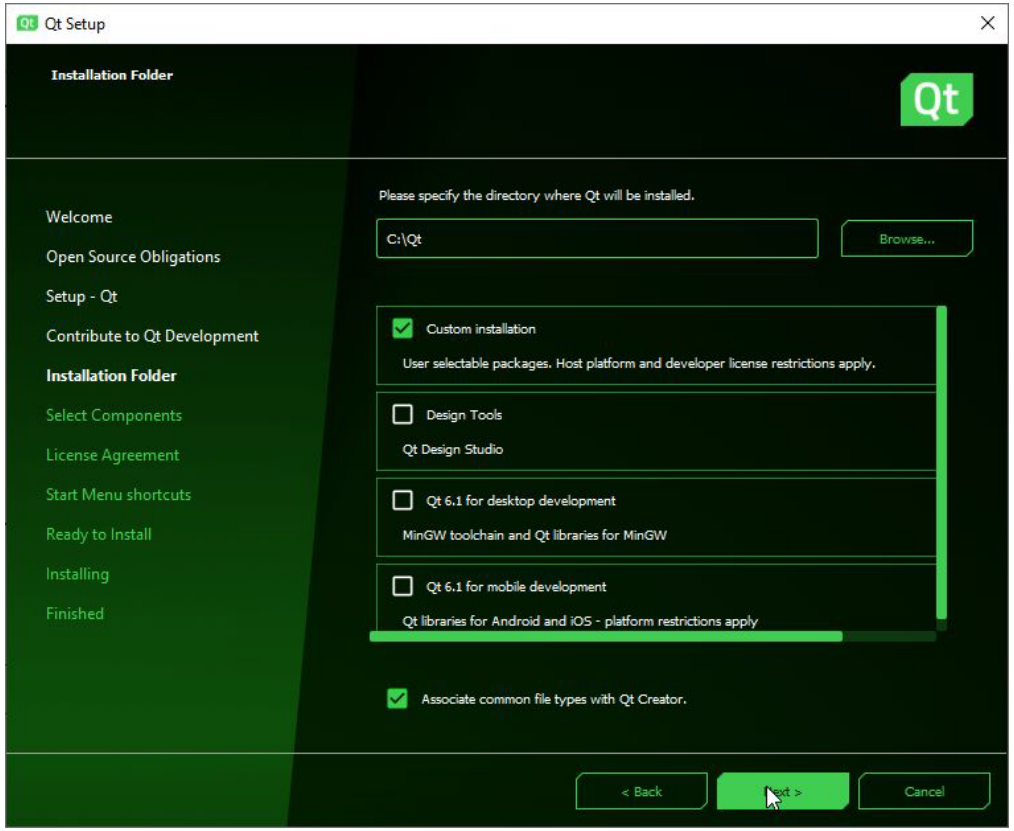
For this course, it's appropriate to install the open source version - but you don't need to download and build from source code. Here is what I did.

1. Go to https://www.qt.io/download-qt-installer and download the installer. It will detect your OS and so on, and download the proper installer for you.
2. 2 - Run the .exe; you should see something like this:



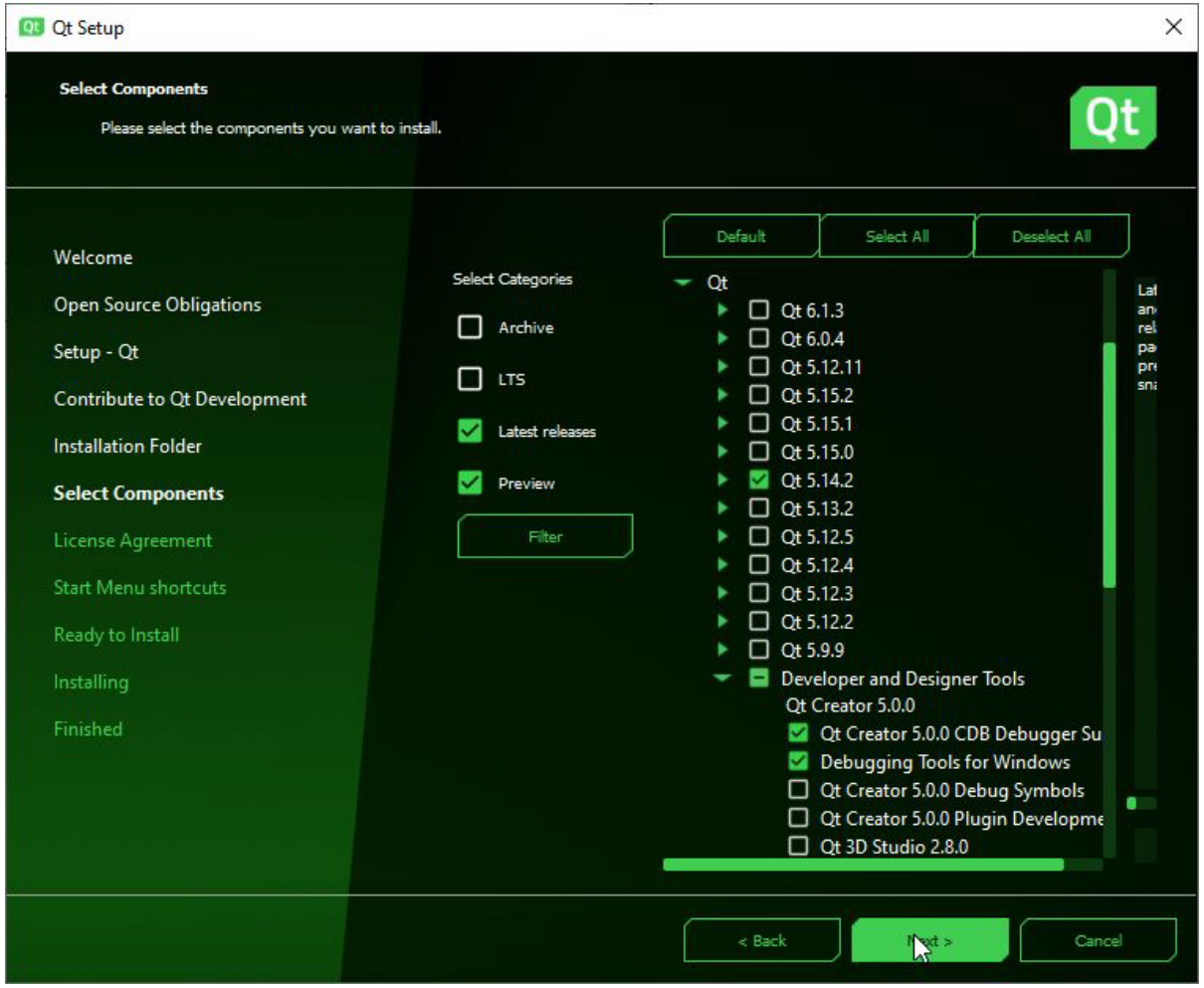
If you happen to have a Qt account, use it here. Otherwise, create one and use the credentials.

3. You have to agree to the GPL license terms and then you will be presented with something like this:



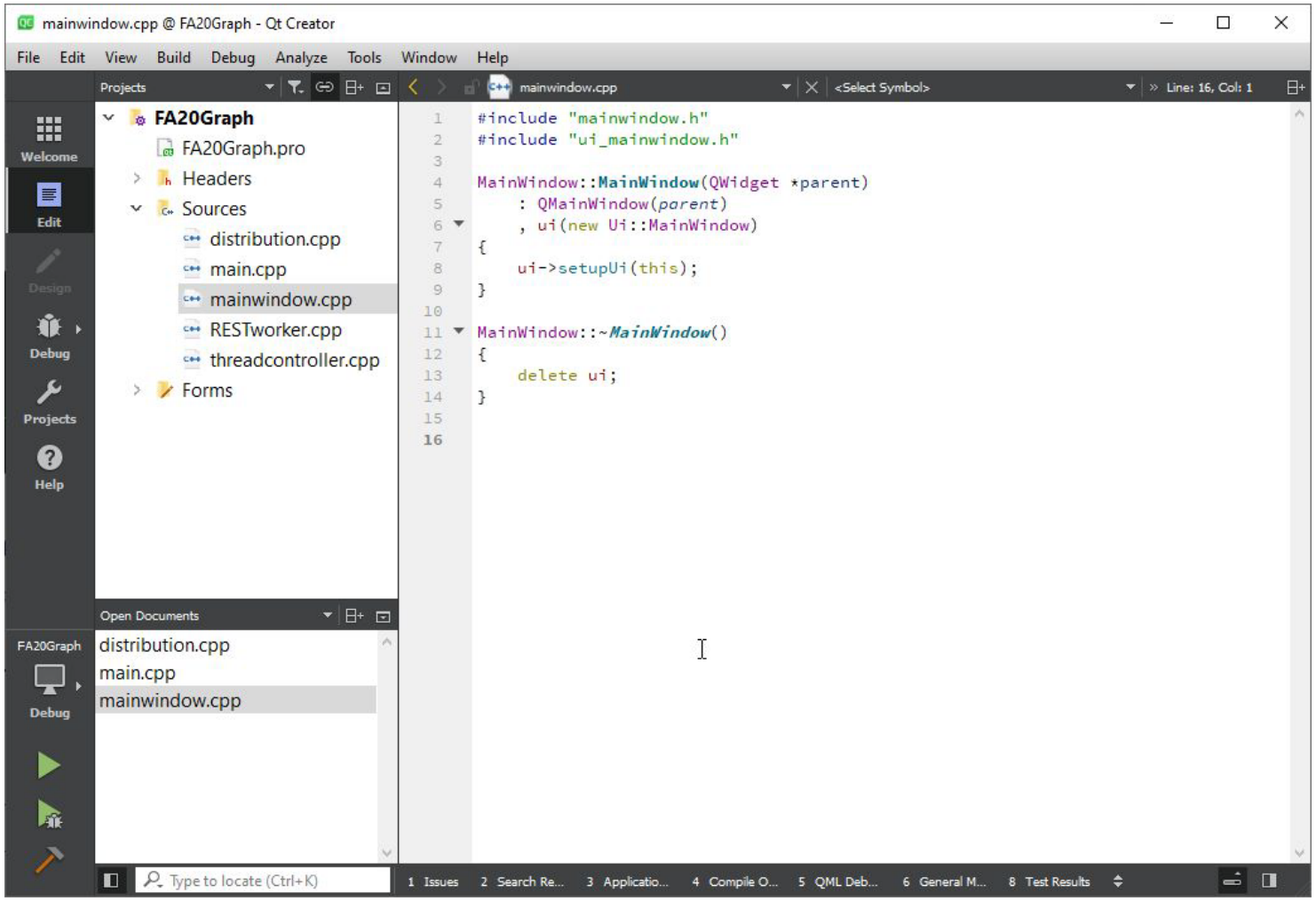
Set the installation directory as appropriate (the Qt installation is LARGE, so choose carefully) and then select Next.

4. On the following page, expand the Qt line and select one of the versions - I use 6.4.0.



Select Next and proceed with installation.

5. The Qt IDE should end up looking something like this...



It's a full-featured C++ development environment. I know that some people use Qt within Visual Studio, but I have never done that and this works well for me.

UNIFIED MODELING LANGUAGE

Unified Modeling Language is a notation for representing software programs of all sizes – it's a fundamental of the design process

- Developed by Booch, Jacobson and Rumbaugh, and others
- Supports four reasons for doing modeling:
  - testing or simulating physical entities before building them;
  - communication with customers;
  - visualization;
  - reduction of complexity.

UML (Unified Modeling Language) 2.0 is widely used for software design and documentation

Structure diagrams:

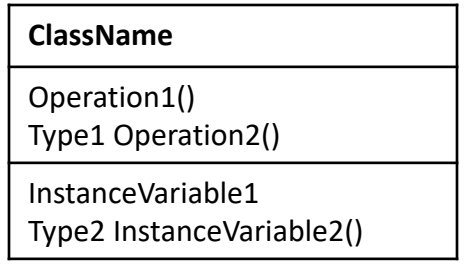
- Class diagrams: names and members of key classes; essential
- Component diagrams: multi-class chunks with well-defined interfaces
- Package diagrams: high-level pieces and their dependencies
- Object diagrams: run-time operation of instances of classes (sometimes called instance diagrams)
- Composite structure diagrams: internal class structure; what is composed of what
- Deployment diagrams: processes overlaid on hardware

Behavior diagrams:

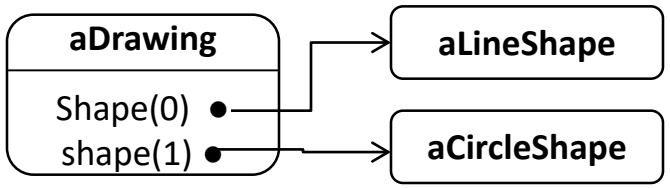
- Activity diagrams: flow of program logic and business rules
- Communication diagrams: how objects call other objects at run-time (collaboration diagrams)
- Sequence diagrams: object activation and messaging
- State machine diagrams: states of objects and events that cause transitions
- Interaction overview diagrams: generally show interactions between activity diagrams
- Timing diagrams: object states and activations in time
- Use case diagrams: interactions between the system and users, the environment and other systems

UML includes many types of diagrams to represent different "views" of the structure and activity of a software entity

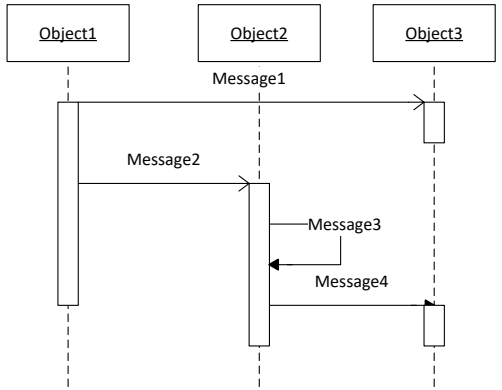
- A class diagram shows classes, their structure and relationships between them;



- An object diagram shows a particular object structure at run-time;



- An interaction diagram shows how requests move between different objects.



UML doesn't look like a "language", but it is – a language for creating and documenting *system design*

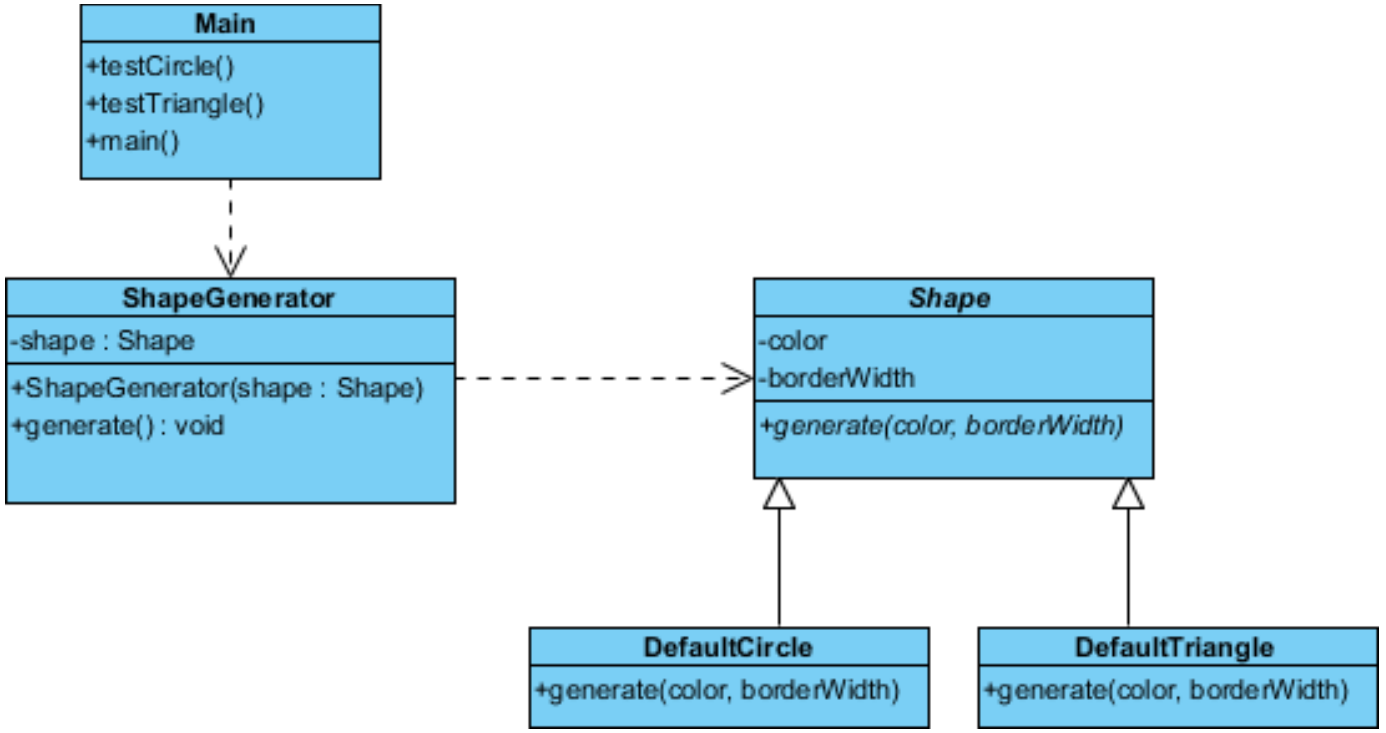
- Some of the drawings are quite useful for documenting specific portions of a system
  - Component diagrams are good for overall design
  - Class diagrams lead naturally to implementations
  - Interaction diagrams and state diagrams are good for elements where sequence and timing are important

- Don't use it for all portions of a system
- Don't take the focus away from writing useful deliverable code
  - Remember the agile principles

UML DIAGRAMS

Diagram	Description	Learning Priority
Class Diagram	Shows a collection of static model elements such as classes and types, their contents, and their relationships. See UML Class diagram guidelines.	High
Component Diagram	Depicts the components that compose an application, system, or enterprise. The components, their interrelationships, interactions, and their public interfaces are depicted. See UML Component diagram guidelines.	High
Package Diagram	Shows how model elements are organized into packages as well as the dependencies between packages. See Package diagram guidelines.	Medium
Object Diagram	Depicts objects and their relationships at a point in time, typically a special case of either a class diagram or a communication diagram.	Low
Composite Structure Diagram	Depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system.	Low
Deployment Diagram	Shows the execution architecture of systems. This includes nodes, either hardware or software execution environments, as well as the middleware connecting them. See UML Deployment diagram guidelines.	Medium
Activity Diagram	Depicts high-level business processes, including data flow, or to model the logic of complex logic within a system. See UML Activity diagram guidelines.	High
Communication Diagram	Shows instances of classes, their interrelationships, and the message flow between them. Communication diagrams typically focus on the structural organization of objects that send and receive messages. Formerly called a Collaboration Diagram. See UML Collaboration diagram guidelines.	Low
Sequence Diagram	Models the sequential logic, in effect the time ordering of messages between classifiers. See UML Sequence diagram guidelines.	High
State Machine Diagram	Describes the states an object or interaction may be in, as well as the transitions between states. Formerly referred to as a state diagram, state chart diagram, or a state-transition diagram. See UML State chart diagram guidelines.	Medium
Interaction Overview Diagram	A variant of an activity diagram which overviews the control flow within a system or business process. Each node/activity within the diagram can represent another interaction diagram.	Low
Timing Diagram	Depicts the change in state or condition of a classifier instance or role over time. Typically used to show the change in state of an object over time in response to external events.	Medium
Use Case Diagram	Shows use cases, actors, and their interrelationships. See UML Use case diagram guidelines.	High

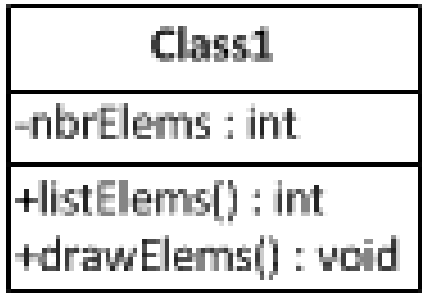
Class Diagram shows members of and relationships between classes



- Inheritance, dependency and containment are all shown by the connections

Each class in a Class Diagram has three sections: class name, methods and member variables

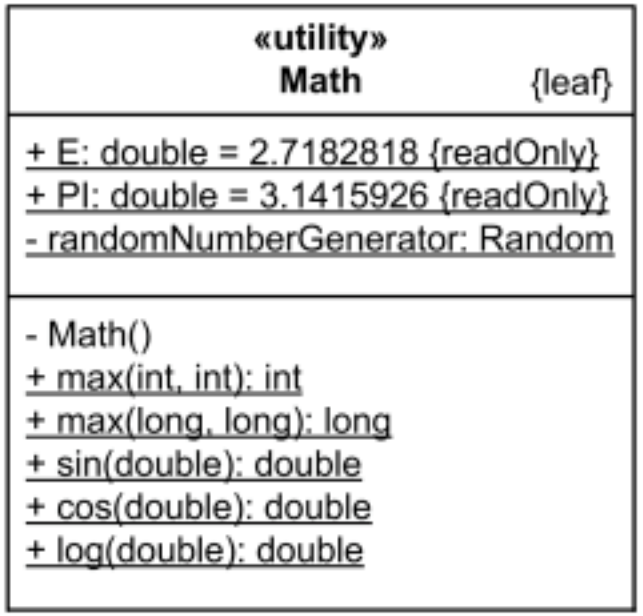
- Class name (italics indicates *abstract class*)
- Key operations (italics for virtual functions)

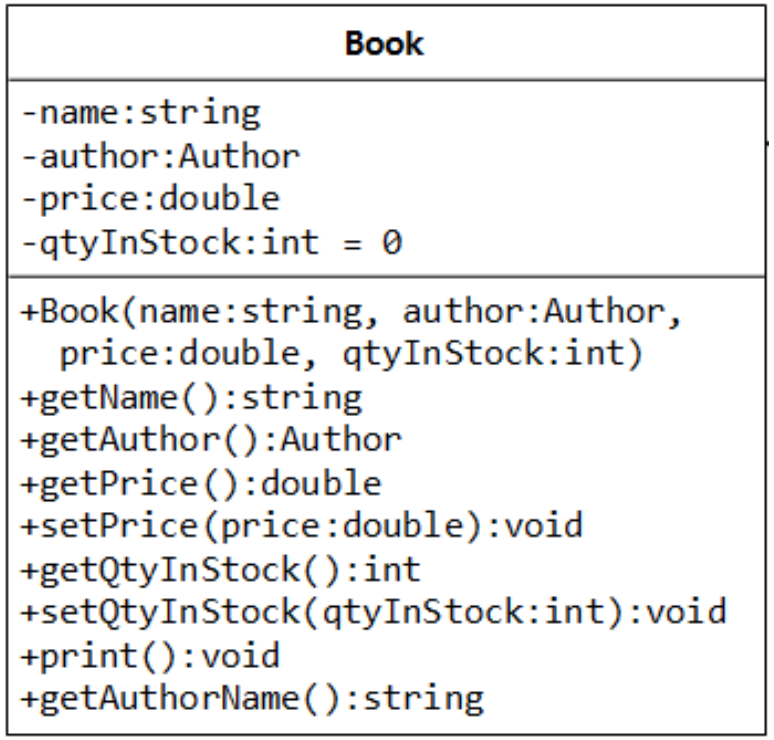


Visio

- Key data members
- Type information is optional but a good idea
- I often use "-" to indicate private members

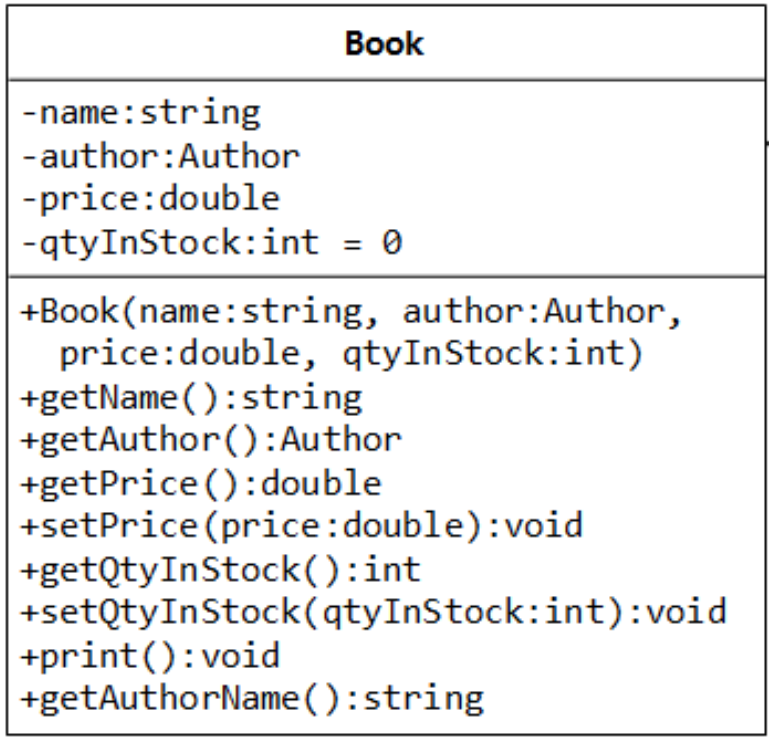
Class diagrams can show more complex class attributes such as return values, static data members and default values



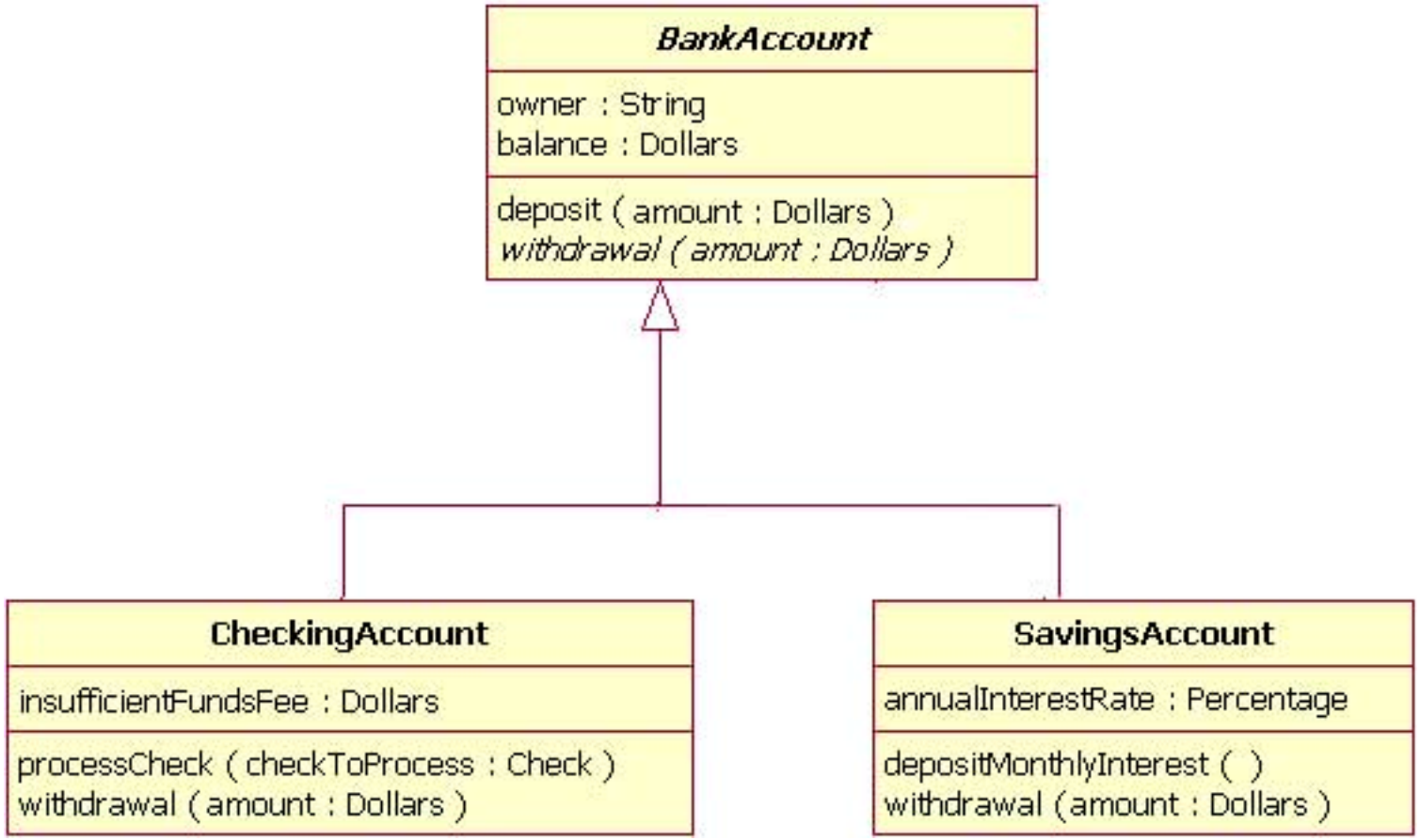


Remember how the class diagram relates to the source code for the class (look at Book.h)

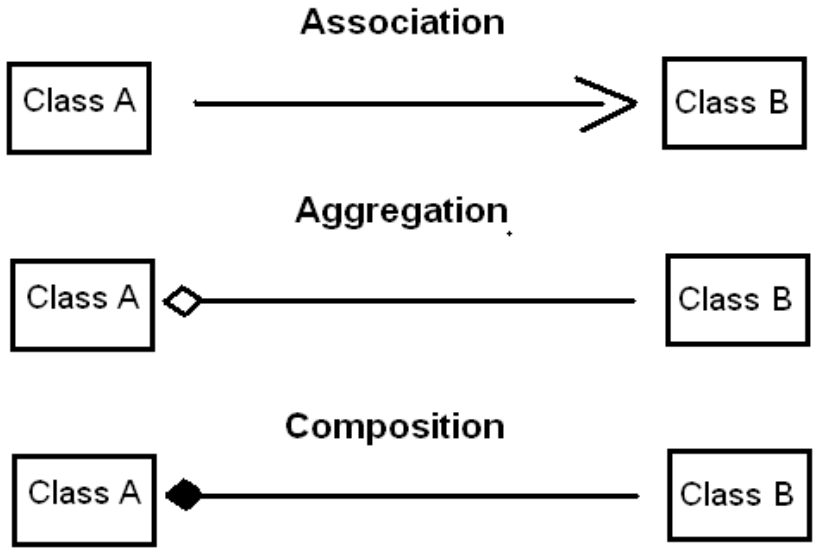
```
class Book {
private:
    std::string name;
    Author author;
    double price;
    int qtyInStock = 0;

public:
    Book(string nm, Author at, double pr, int q);
    string getName() { return name; }
    Author getAuthor() { return author; }
    double getPrice() { return price; }
    void setPrice(double pr);
    int getQtyInStock() { return qtyInStock; }
    void setQtyInStock(int qt);
    void print();
    string getAuthorName() { return author.getName(); }
```



Inheritance is shown on a class diagram by a large hollow arrow



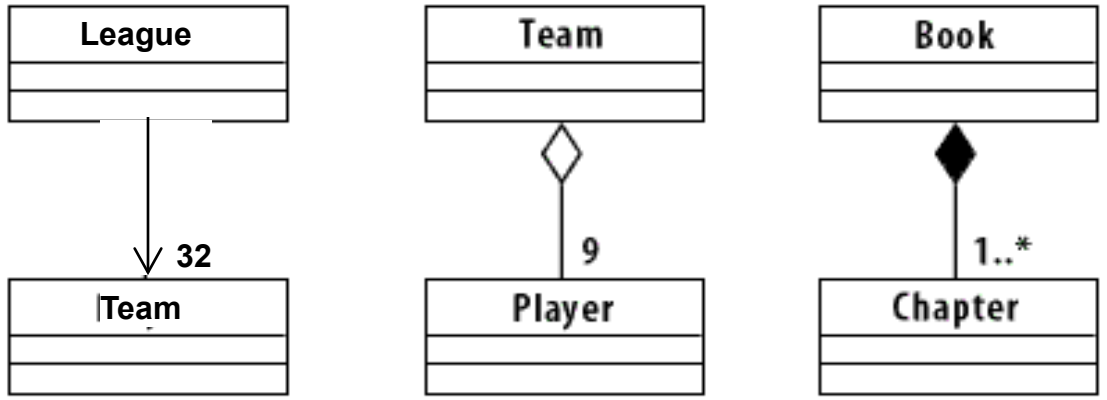
A full class diagram is used to show important relationships (or dependencies) between classes: Association, Aggregation and Composition



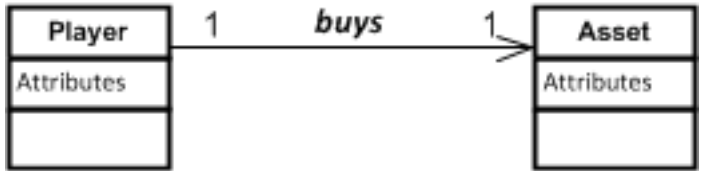
- Association means that class A contains a reference to an object of type B
- Aggregation means that class A creates a variable of type B
  - Often considered as the same as association
- Composition indicates that A contains a variable of type B

Association, Aggregation and Composition can be confusing

	Association	Aggregation	Composition
Owner	No owner	Single owner	Single owner
Life time	Have their own lifetime	Have their own lifetime	Owner's life time
Child object	Child objects all are independent	Child objects belong to a single parent	Child objects belong to a single parent



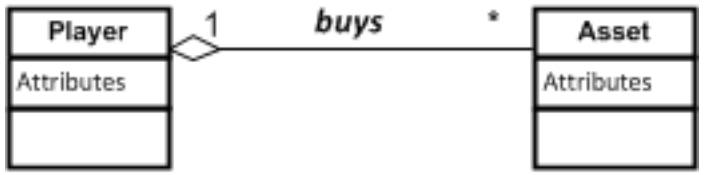
Association means that class A contains a reference to an object of type B




```
class Asset { ... }

class Player {
private:
    Asset* mpAsset;
public:
    Player(Asset* purchasedAsset) {
        mpAsset = purchasedAsset;
    }
}
```

```
public class Asset { ... }

public class Player {
        private Asset asset;
        public Player(Asset purchasedAsset) {
... }
 /*Set the asset via Constructor or a setter*/
}
```

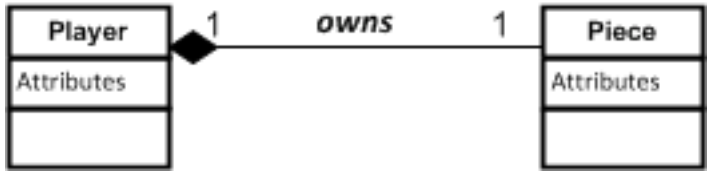
Aggregation means that class A creates a variable of type B




```
class Asset { ... }

class Player {
private:
    Piece* mpPiece;
public:
    Player() {
        this->mpPiece = new Piece();
    }
}
```

```
public class Piece { ... }

public class Player {
    private Piece mPiece;
    public Player() {
        this.mPiece = new Piece();
    }
 // Player is responsible for creating the piece
}
```

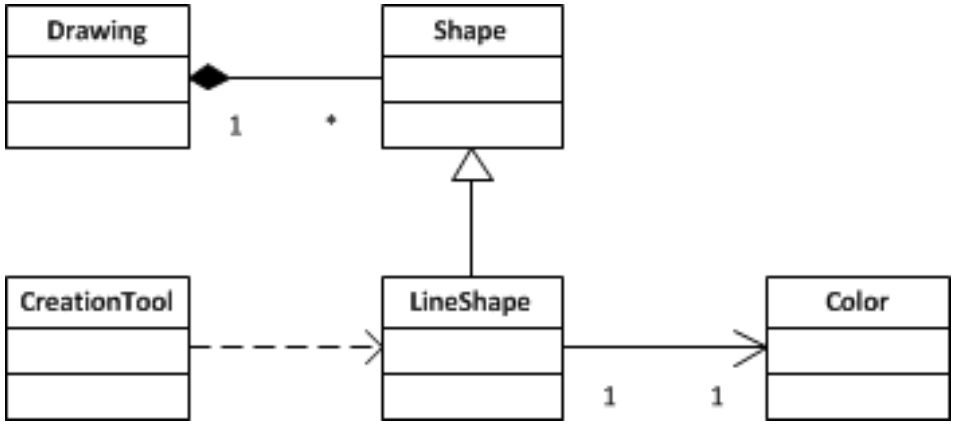
Composition indicates that A contains a variable(s) of type B




```
class Asset { ... }

class Player {
private:
    std::vector<Asset> assets;
public:
    void addAsset(Asset purchasedAsset) {
        assets.push_back(Asset(purchasedAsset));
// note use of copy constructor
    }
}
```

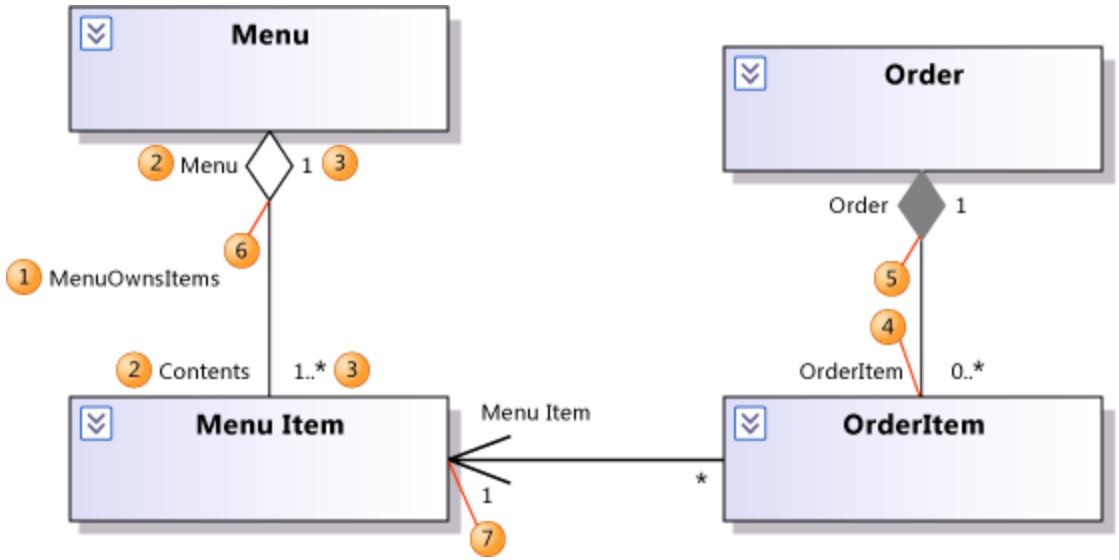
```
public class Asset { ... }

public class Player {
    private List assets;
    public Player(Asset purchasedAsset) {
        assets.Add(purchasedAsset);
    }
}
```

Class diagrams are most useful for showing the relationship between the classes in the design – what's part of what, etc.

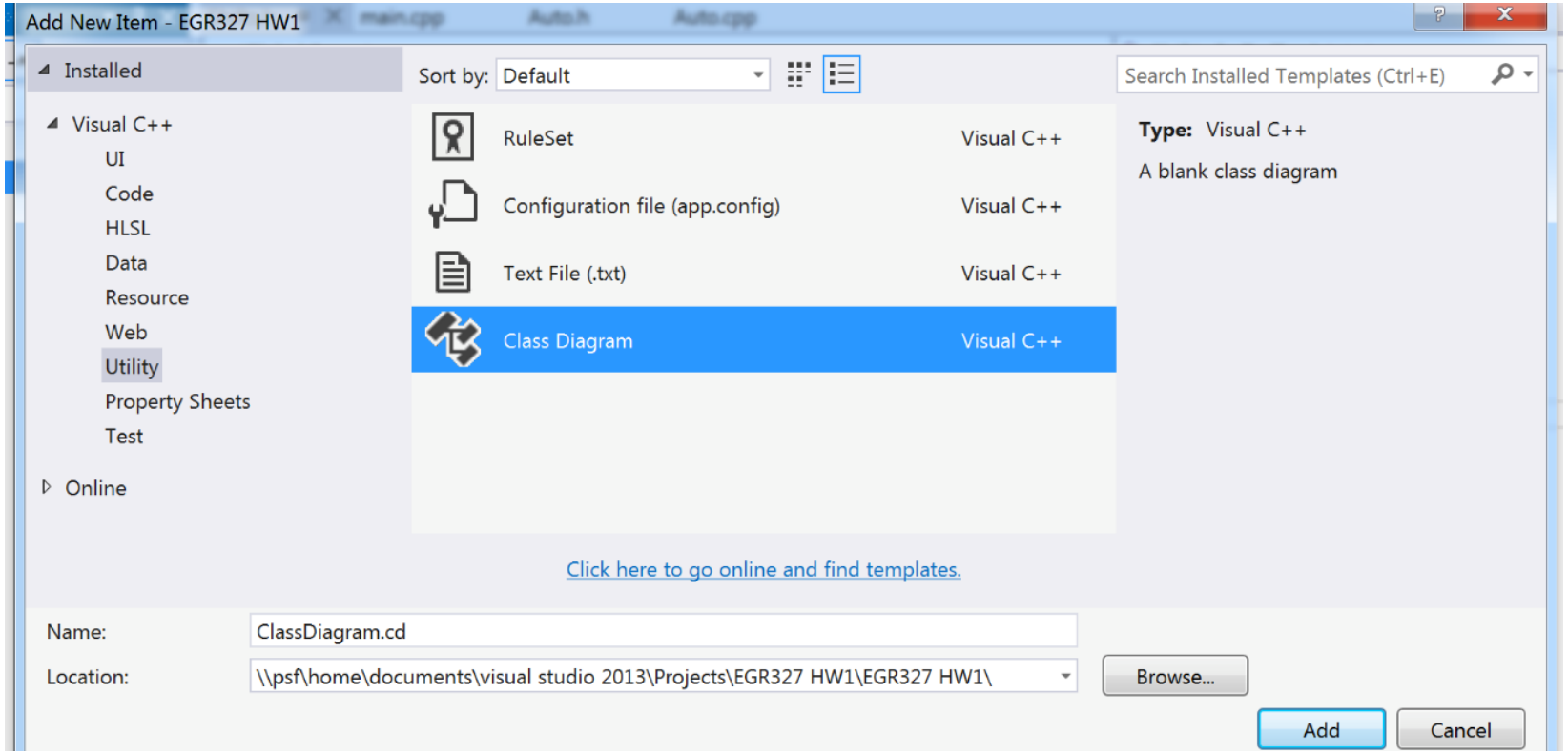


- LineShape inherits from Shape
- Drawing contains Shapes
  - zero or more – could be 1, 1..*, 0..*, …)
- CreationTool instantiates (creates) LineShapes
- LineShape contains a reference to an external Color

Property	Default	Description
Role Name (2)	Name of the type at this role	The name of the role. Appears near the end of the association on the diagram.
Aggregation	None	**None** (4) – represents a general relationship between instances of the classes. **Composite** (5) – the object at this role contains the object at the opposite role. **Shared** (6) – object at this role contains references to the object at the other role.
Is Navigable	True	The association can be read in this direction. Given an instance of the opposite role, the software that you are describing can efficiently determine the associated instance in this role. If one role is Navigable and the other is not, an arrow appears (7) on the association in the navigable direction.
Multiplicity (3)	1	**1** – this end of the association always links to one object. In the figure, every Menu Item has one Menu. **0..1** – either this end of the association links to one object, or there is no link. * – every object at the other end of the association is linked to a collection of objects at this end, and the collection may be empty. **1..*** – every object at the other end of the association is linked to at least one object at this end. In the figure, every Menu has at least one Menu Item. *n..m* – each object at the other end has a collection of between *n* and *m* links to objects at this end.
Visibility	Public	Public – visible globally Private – not visible outside the owning type Protected – visible to types derived from the owner Package – visible to other types within the same package.

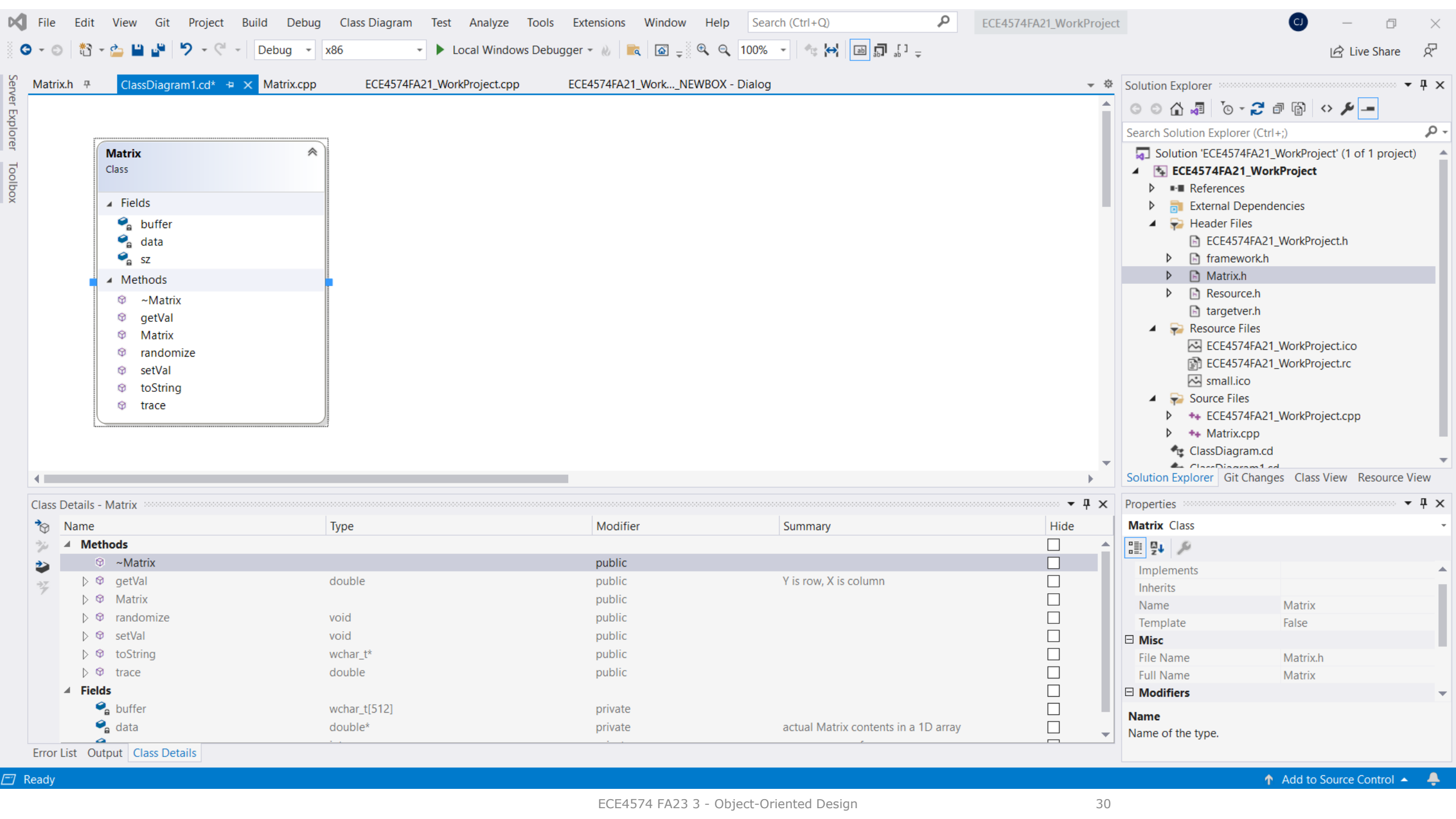


There are a number of tools for creating class diagrams…

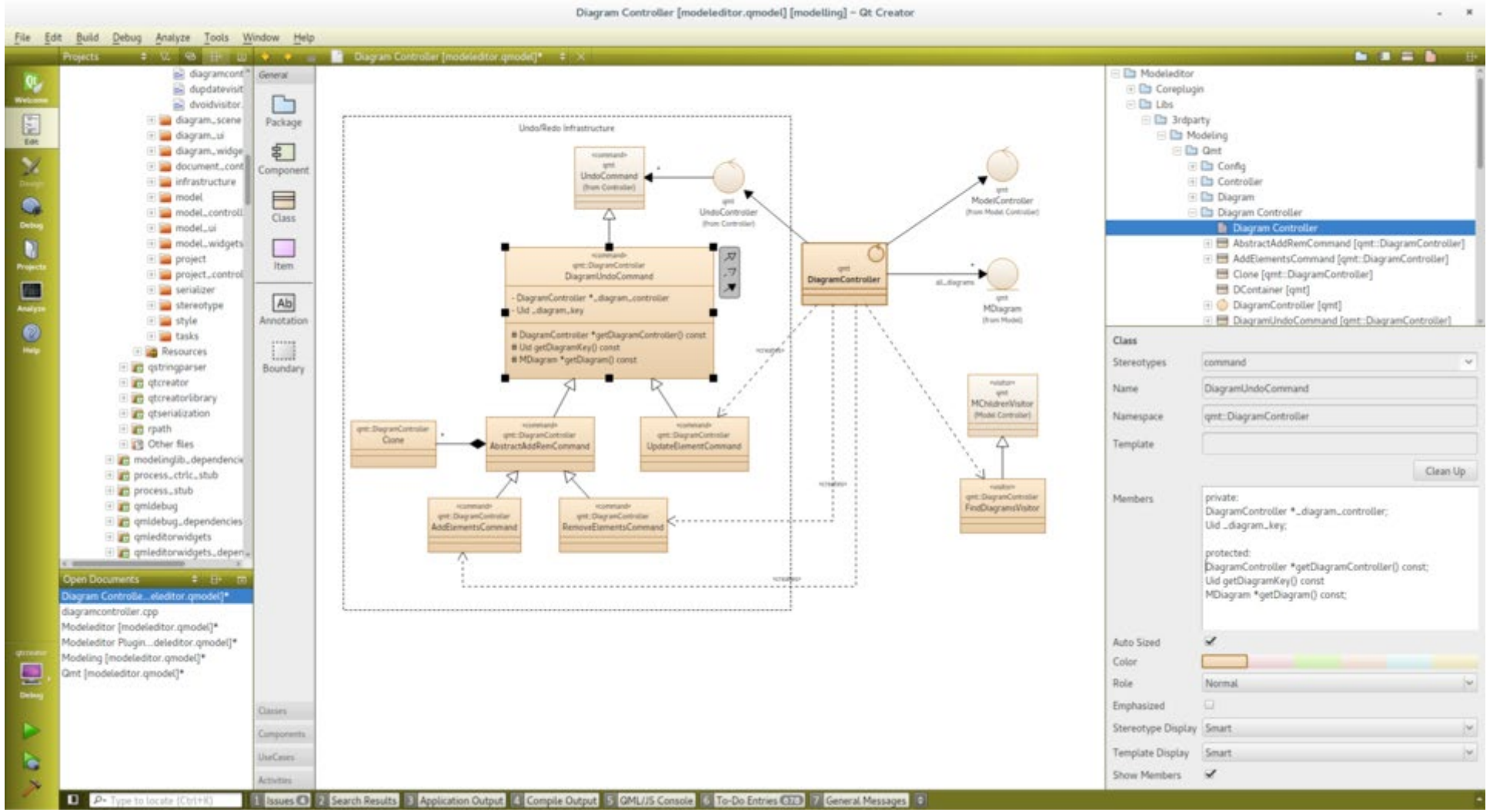


You can generate class diagrams for existing C++ code using Visual Studio

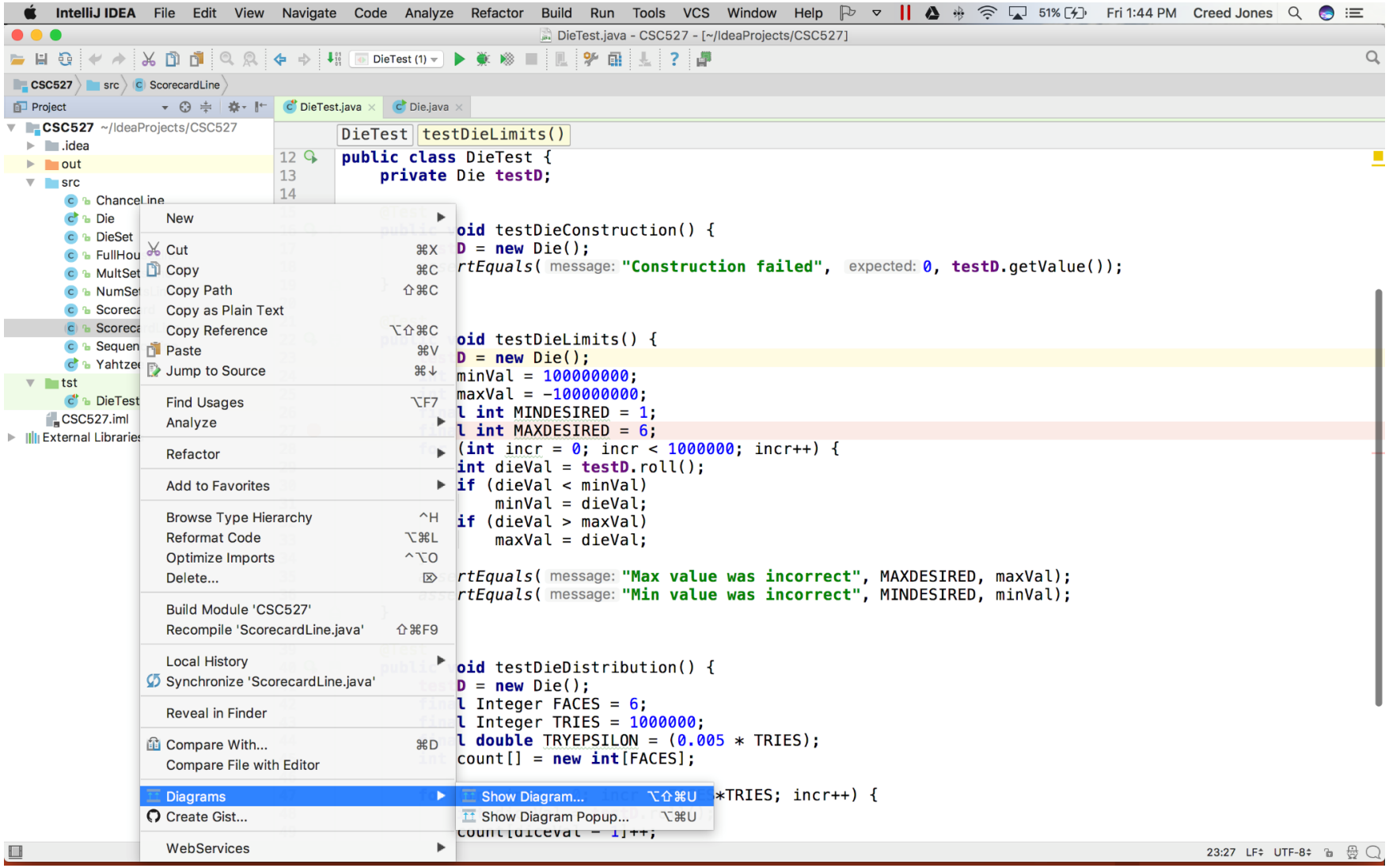
- Install the Class Designer
  - https://docs.microsoft.com/en-us/visualstudio/ide/class-designer/how-to-add-class-diagrams-to-projects?view=vs-2019
- Right click on the class or project you want a diagram of
- The tool will generate the class diagram


File Edit View Git Project Build Debug Class Diagram Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
ECE4574FA21_WorkProject
Debug
x86
Local Windows Debugger
100%
Live Share
Matrix.h
ClassDiagram1.cd*
Matrix.cpp
ECE4574FA21_WorkProject.cpp
ECE4574FA21_Work..._NEWBOX - Dialog
Matrix
Class
Fields
buffer
data
sz
Methods
~Matrix
getVal
Matrix
randomize
setVal
toString
trace
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ECE4574FA21_WorkProject' (1 of 1 project)
ECE4574FA21_WorkProject
References
External Dependencies
Header Files
ECE4574FA21_WorkProject.h
framework.h
Matrix.h
Resource.h
targetver.h
Resource Files
ECE4574FA21_WorkProject.ico
ECE4574FA21_WorkProject.rc
small.ico
Source Files
ECE4574FA21_WorkProject.cpp
Matrix.cpp
ClassDiagram.cd
Solution Explorer
Git Changes
Class View
Resource View
Class Details - Matrix
Name
Type
Modifier
Summary
Hide
Methods
~Matrix public
getVal double public Y is row, X is column
Matrix public
randomize void public
setVal void public
toString wchar_t* public
trace double public
Fields
buffer wchar_t[512] private
data double* private actual Matrix contents in a 1D array
Properties
Matrix Class
Implements
Inherits
Name Matrix
Template False
Misc
File Name Matrix.h
Full Name Matrix
Modifiers
Name
Name of the type.
Error List
Output
Class Details
Ready
Add to Source Control

Qt has a ModelEditor plugin – note that it's experimental and a little rough around the edges

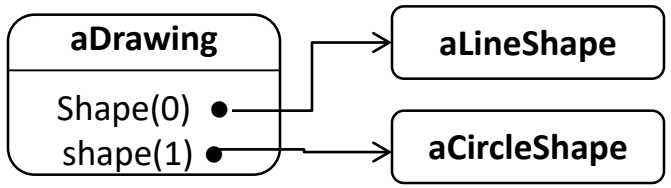


There are a number of tools for creating class diagrams; here's the IntelliJ Java IDE



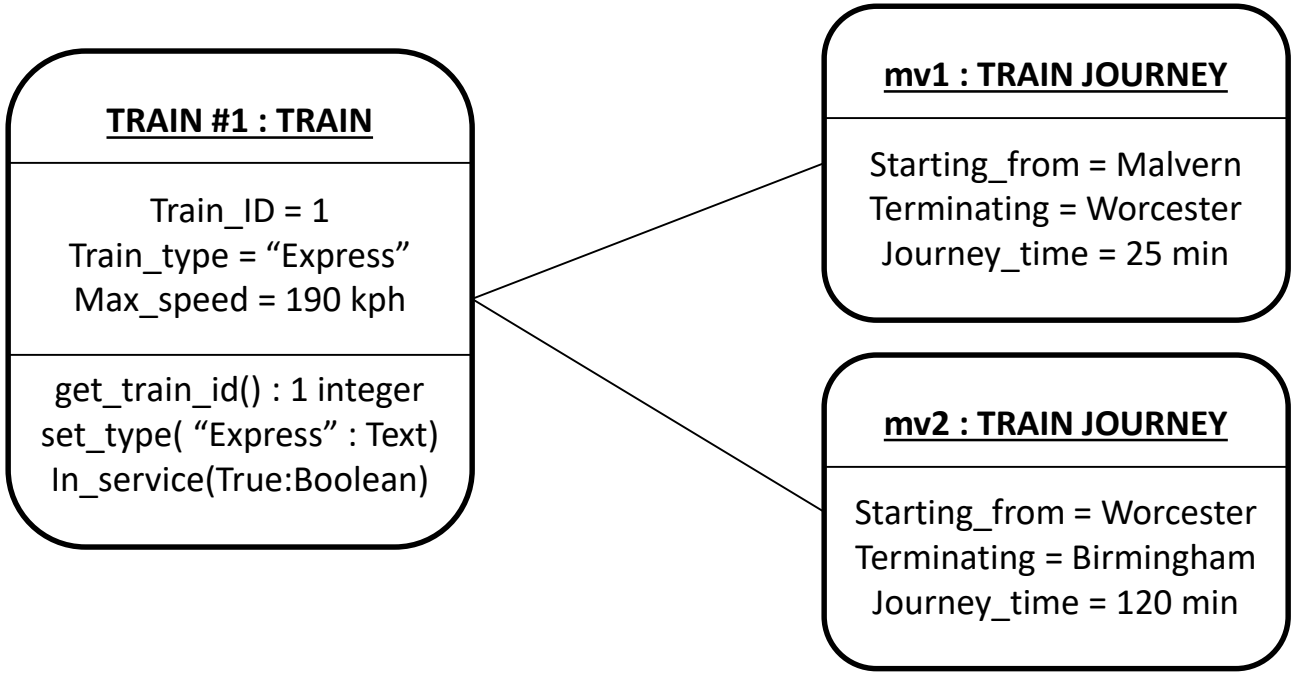
Object Diagrams show a snapshot of objects in the design at some time

- An instance of *Class* is labeled *aClass*
- If an object of class Drawing will contain two shapes, a Line and a Circle, the object drawing will look like:

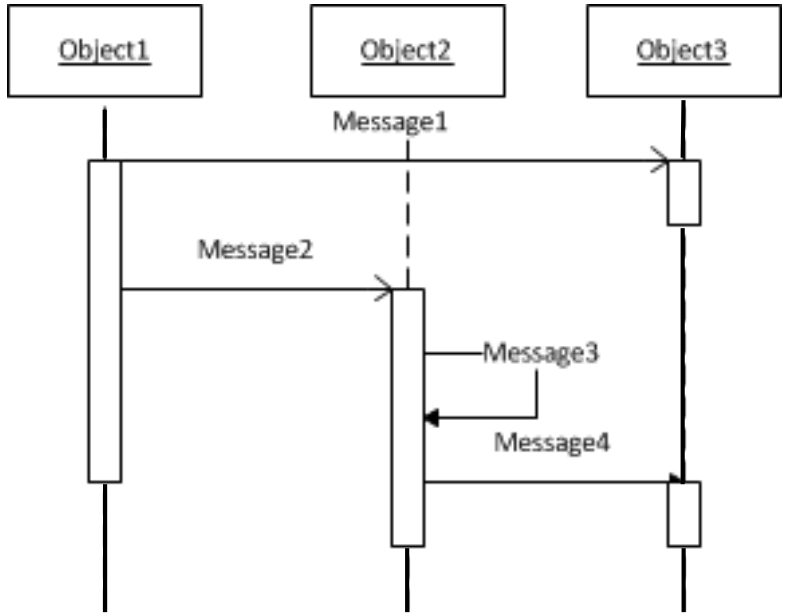


What is an object diagram used for? To represent relationships between actual objects during execution – usually at some interesting time

- Say we want to design/test/document the way that our SW handles multiple train journeys for the same train
- Draw an object diagram – the key is that there are two objects of the same class type

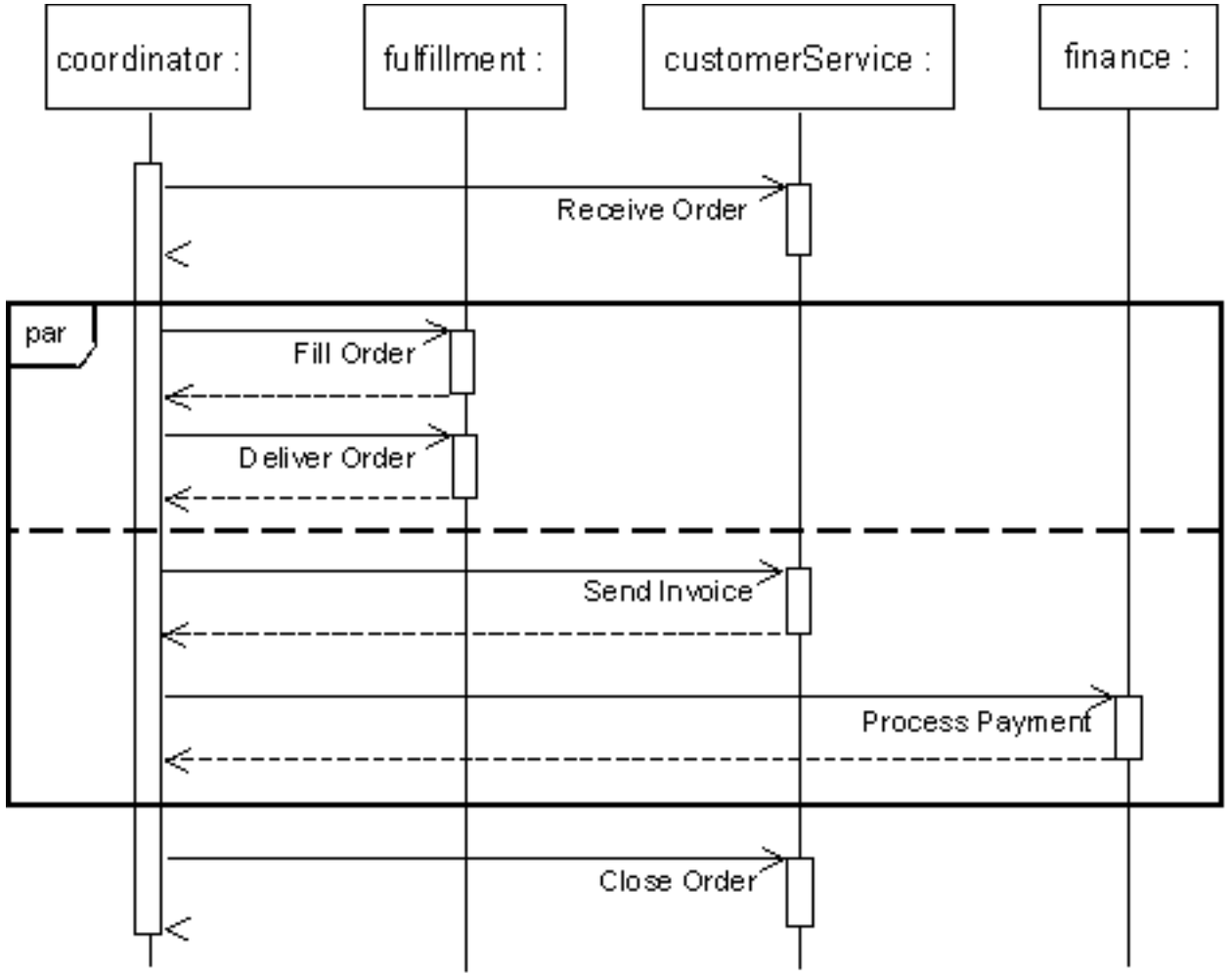


Interaction Diagrams are the best method to show the time-based activity between objects (creation, messaging, etc).

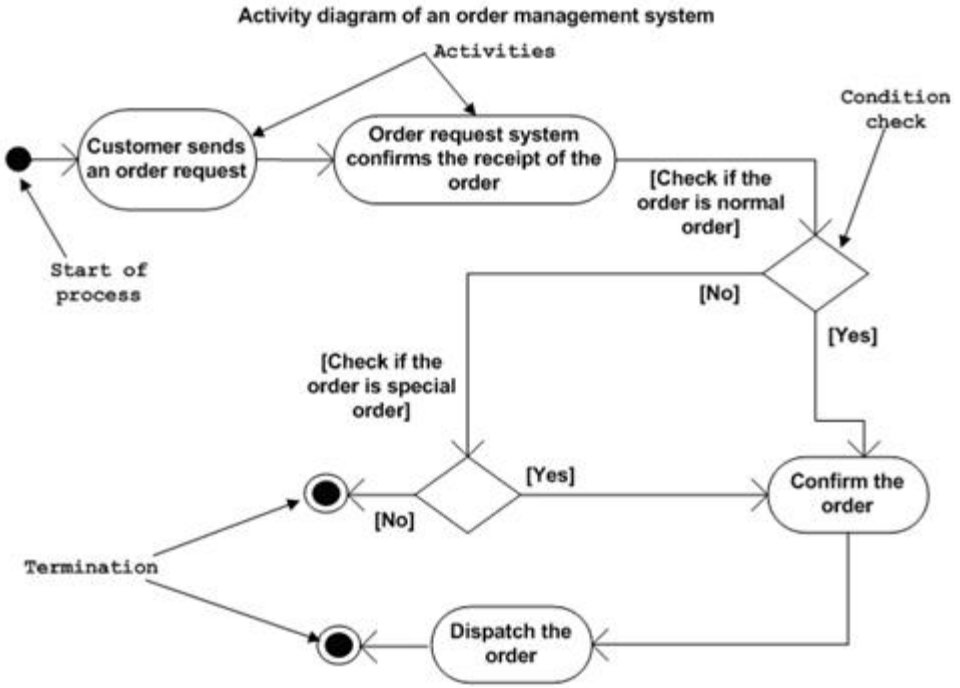


- Vertical lines show the lifetime of each object
  - dotted lines mean not created yet
  - Visio doesn't support this well
- Vertical bars show the *activation* time (when is the object actually running)
- Messages can go either way

Interaction diagram for placing and processing an order

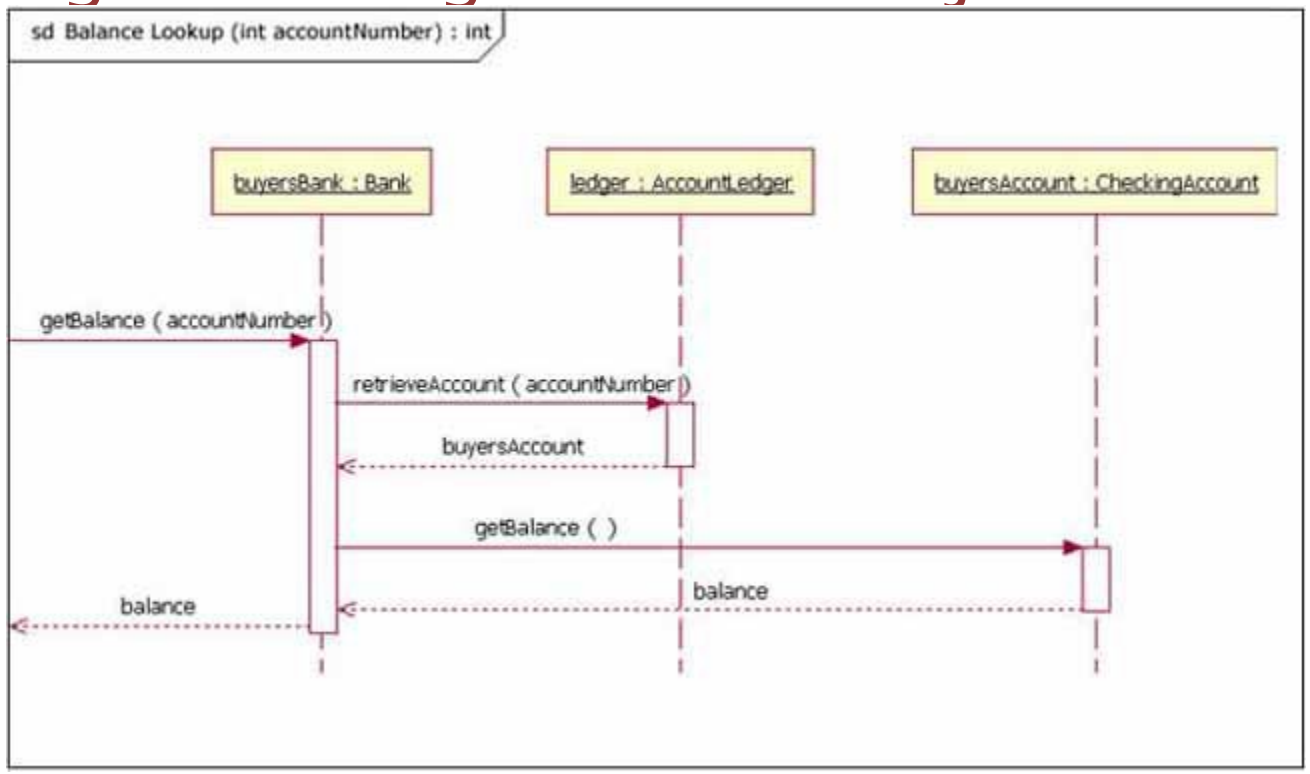


Activity Diagrams show the flow of a program or business process (like a flowchart)



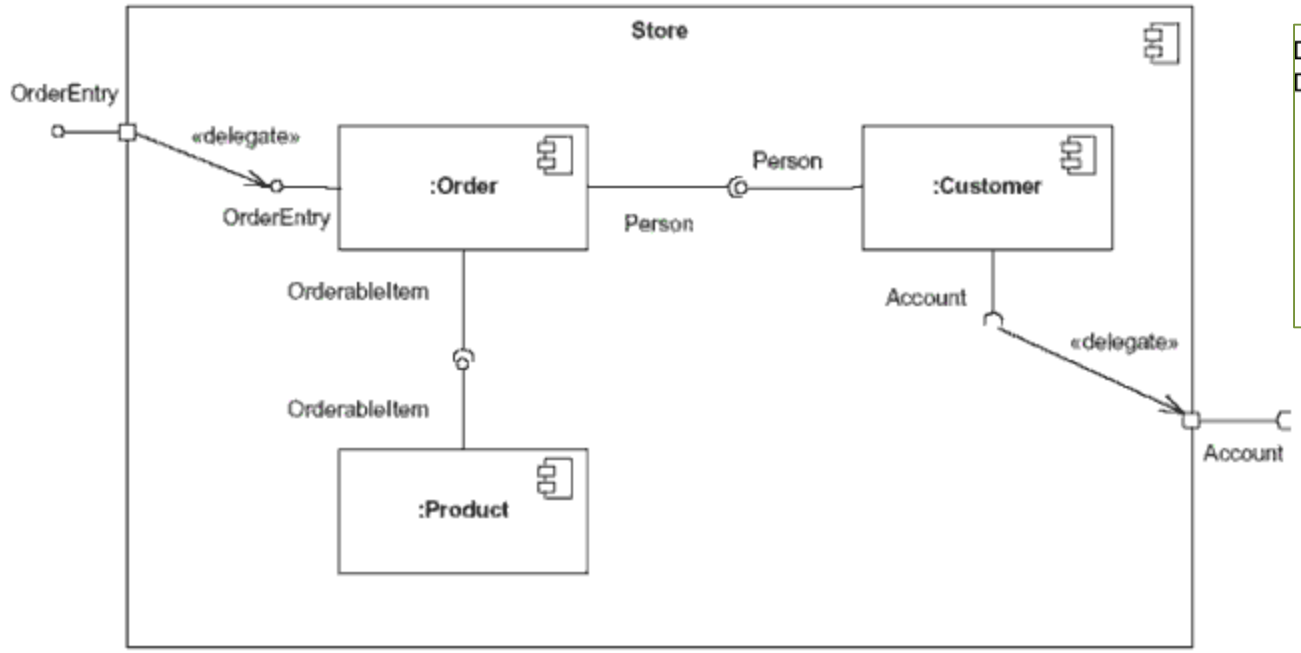
- How is this different than a simple flowchart?
  - we're not describing simple procedural code
  - they can represent concurrent processing

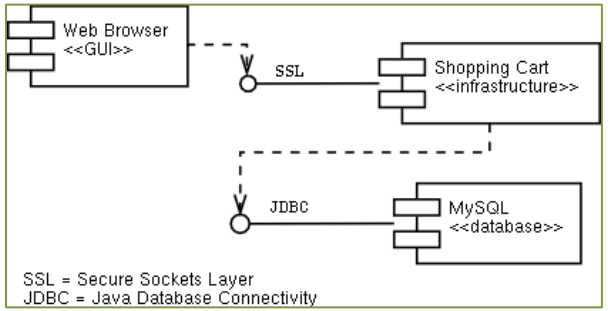
Sequence (or Interaction) Diagrams show control and messages flowing between objects



- They show lifetime, activation and messaging behavior of objects
- Can show synchronous (call/return) and asynchronous (non-blocking) messages

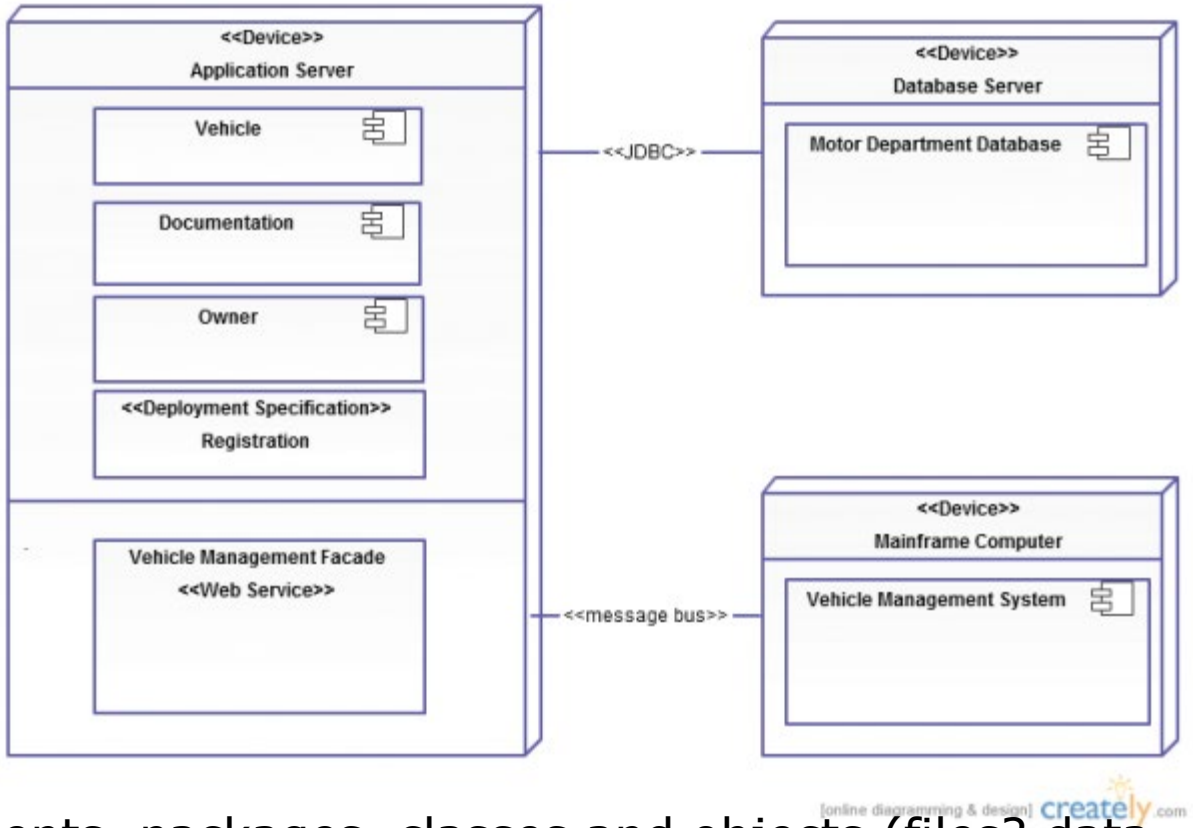
Component Diagrams describe things at a higher level than class diagrams





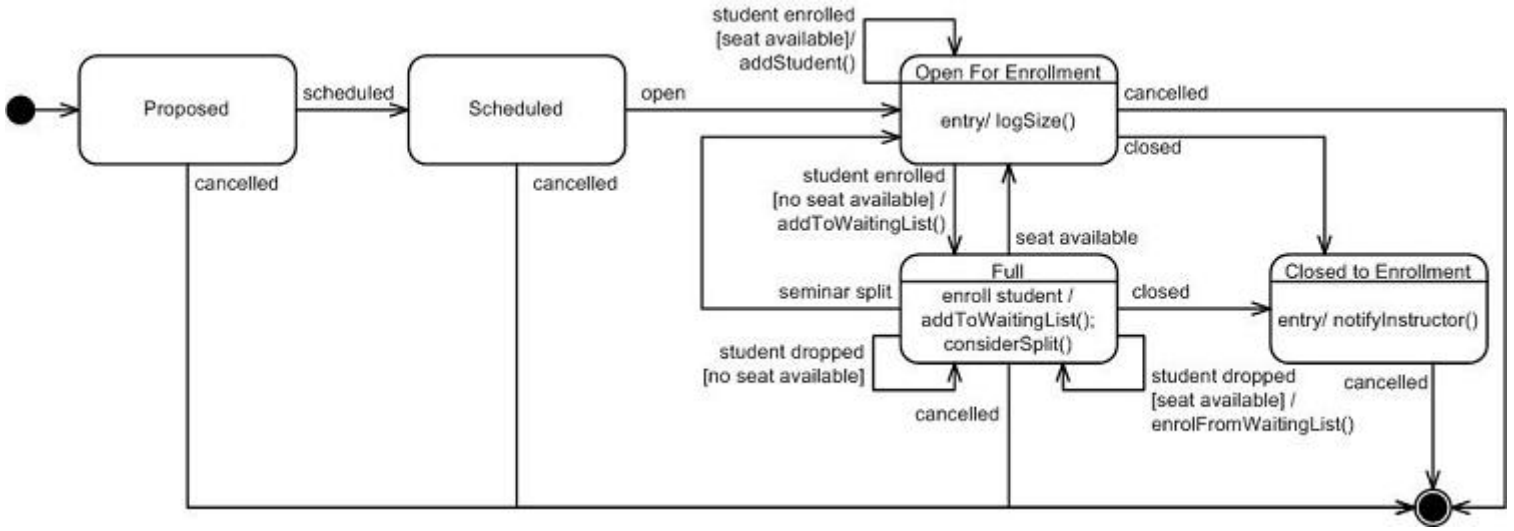
- Interfaces, required connections and external ports are all shown by new interconnection symbols

Deployment Diagrams show the runtime placement of SW components on HW entities



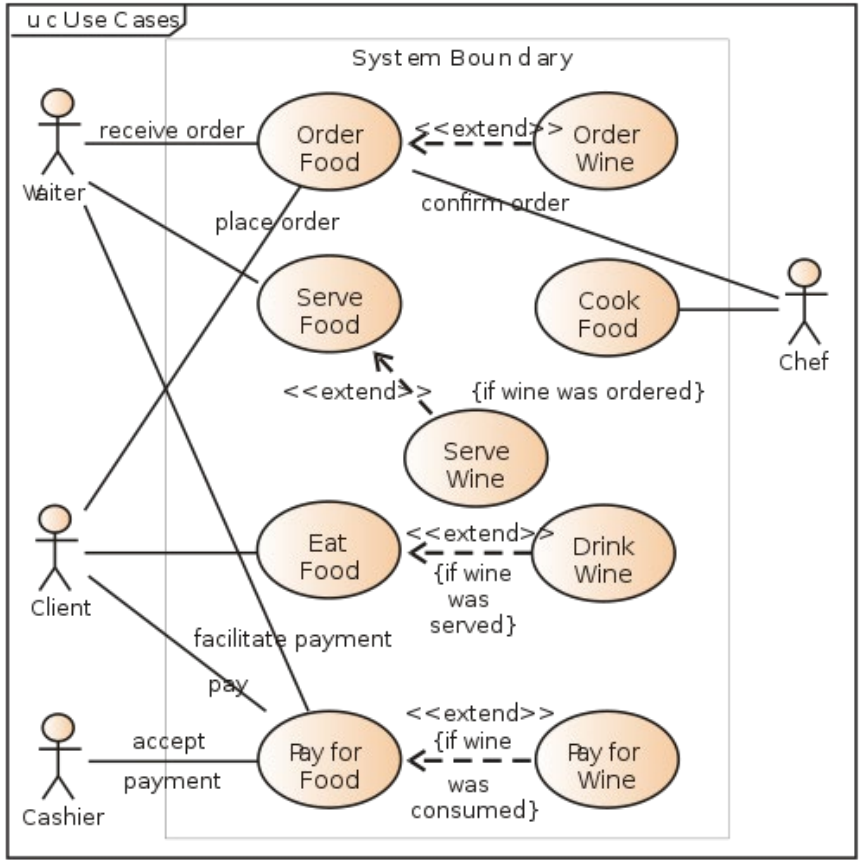
- Components, packages, classes and objects (files? data elements?) can be shown on various processes, hardware platforms or sub-architectures

State Machine Diagrams show how a single object moves from state to state – including stimuli that cause transitions



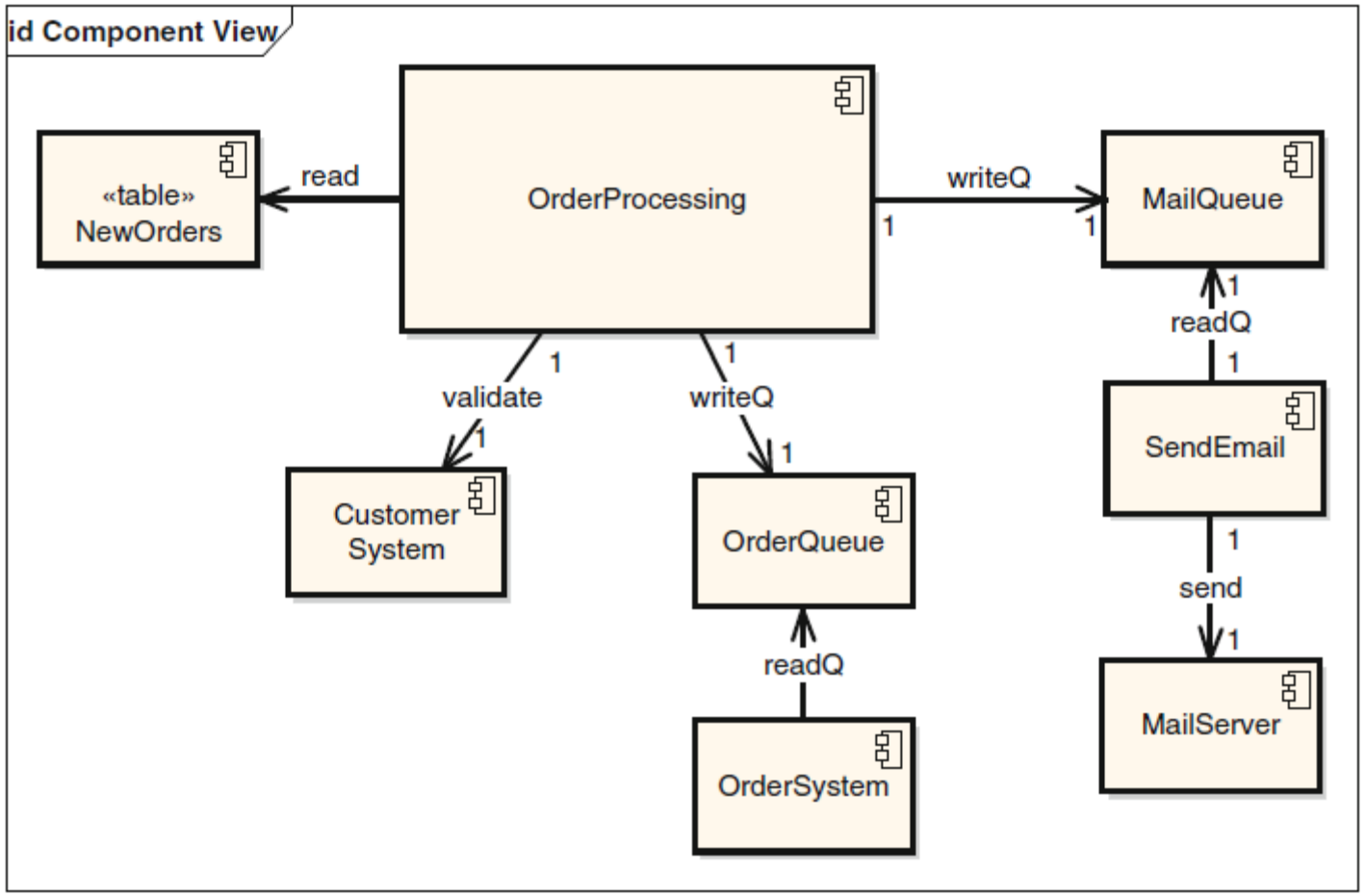
- Some state machines are so complex that they are represented in pieces
  - Move some off-page
  - Collect some states into a "super-state"

Use Case Diagrams show interactions between system and "user" in a given scenario of use

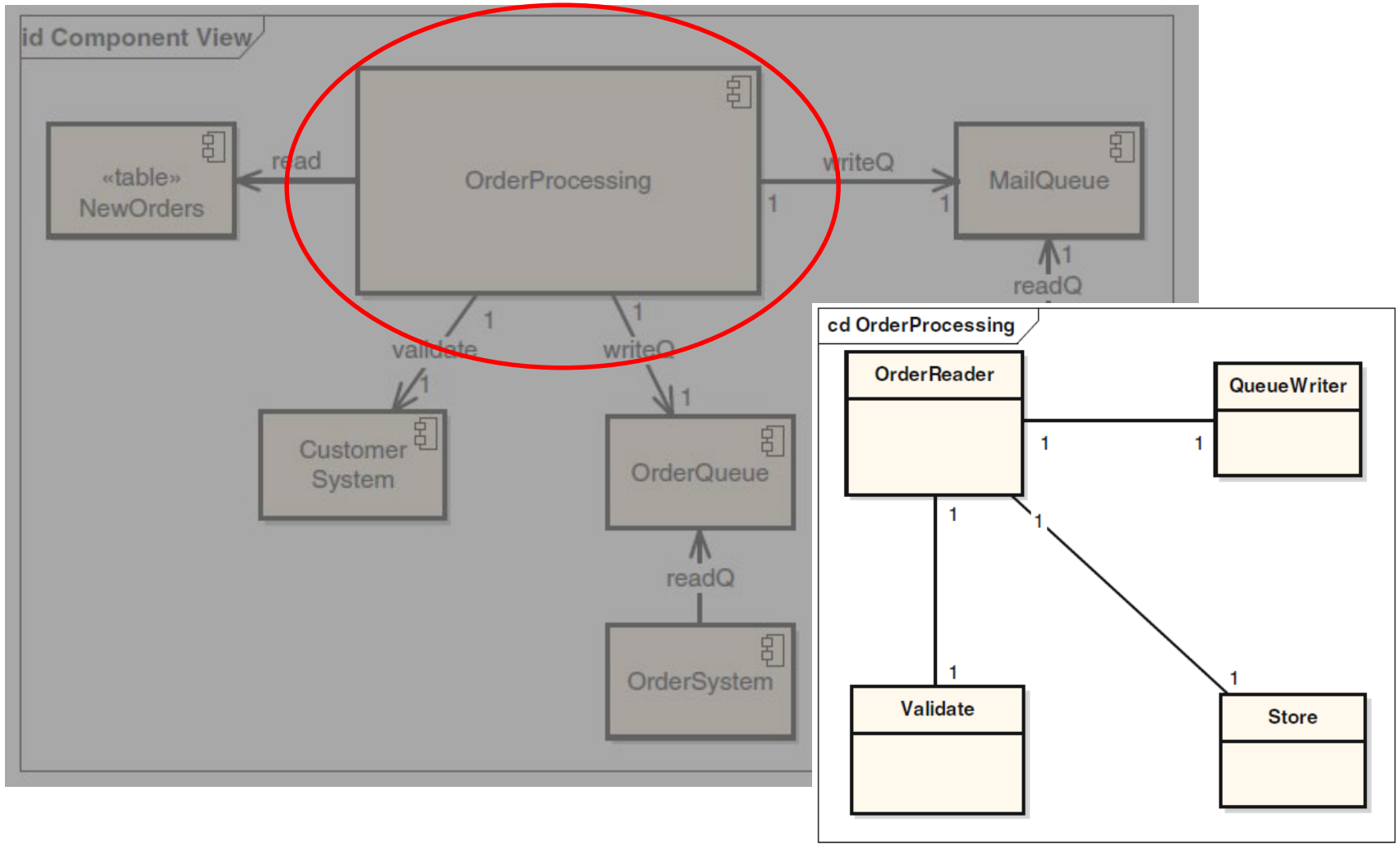


- Not all actors are human
- Note the "extends" notation – may not be part of basic use case

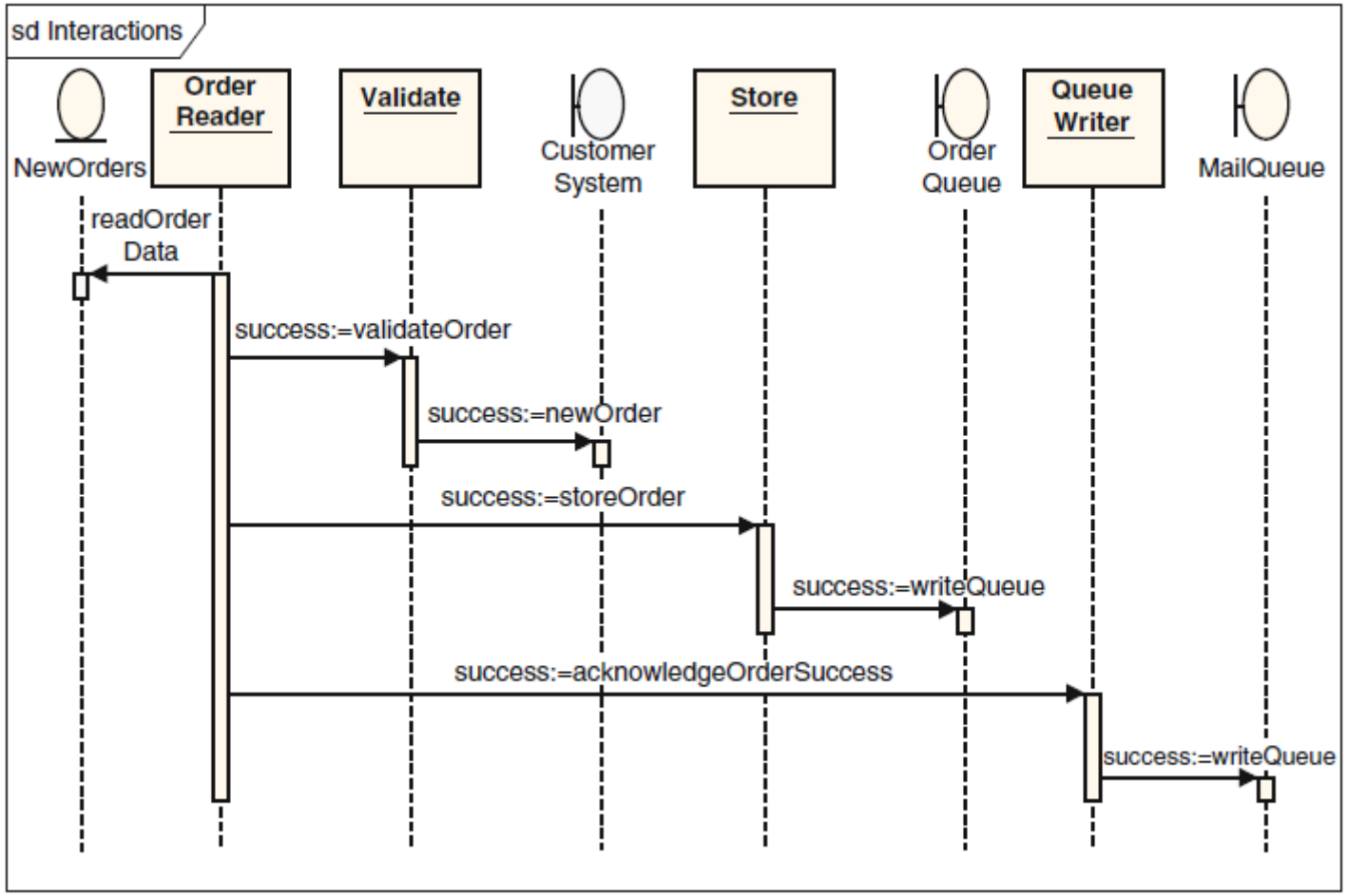
How does all this work together? Consider a sample architecture…



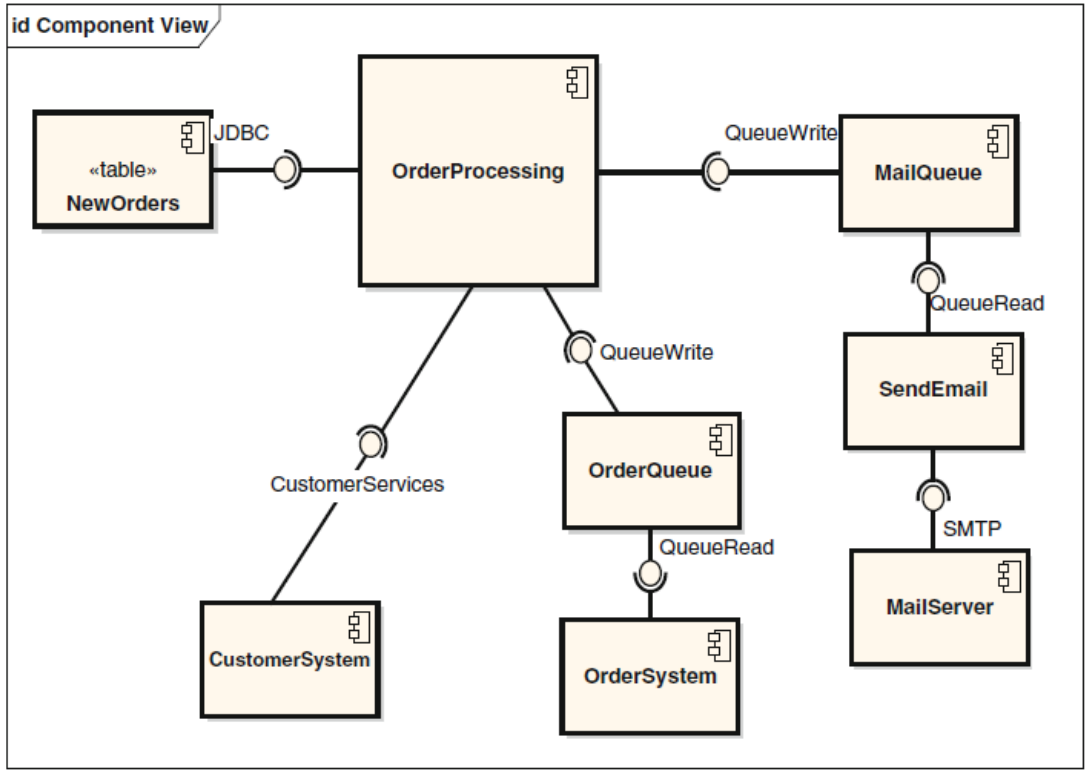
Here is a look inside the OrderProcessing component – in a class diagram



A sequence diagram for the same order processing system



Component diagrams are the place to record interfaces supported and required, data types, messaging styles, names, etc.



- Notate interfaces with key information
- Each component can be a "drill-down" to various class/object/interaction diagrams (as needed)

Topics for Today

Unified Modeling Language

- UML Concept
- UML Diagrams
  - class
  - interaction
  - component
  - use case
- Simple example