OO Design Principles

Contents

[1 Single Responsibility Principle 1](#_Toc146108962)

[2 MVC (Model-View-Controller) Architecture 2](#_Toc146108963)

[3 Example 3](#_Toc146108964)

[3.1 Domain Classes 3](#_Toc146108965)

[3.2 System Classes 4](#_Toc146108966)

[3.3 Starting the System 6](#_Toc146108967)

[3.4 Use Case: Add a Blob 6](#_Toc146108968)

[3.5 Design Principles 8](#_Toc146108969)

[Appendix 1 Resources 8](#_Toc146108970)

# Single Responsibility Principle

There are a number of OO design principles that help make software designs more understandable, flexible, and maintainable, and extensible. [Dr. Robert C. Martin](https://en.wikipedia.org/wiki/Robert_C._Martin) (*Uncle Bob*) proposed five design principles (among others), called the [SOLID](https://en.wikipedia.org/wiki/SOLID) principles.

One import principle, is the Single Responsibility Principle[[1]](#footnote-1)[[2]](#footnote-2)[[3]](#footnote-3)[[4]](#footnote-4). It states:

* A class should have only one responsibility
* A class should have only one reason to change

We say that a class that adheres to the SRP has *high cohesion.* As we develop a software system, classes can become bloated; they take on too much responsibility. If we look carefully at such classes, we can often break them into two (or more) classes each with a single responsibility.

Here is a simple example that violates the single responsibility principle: Suppose we have an Employee class with name, payrate, hours worked, *etc* attributes and a method to calculate their pay. Suppose this class also has a method to write the instance variables to a text file (or database, *etc*.). This is two responsibilities: one is to manage the employee’s attributes and the other is to persist it to disk. We should always have a separate class(es) to do data persistence.

Another example is building a GUI based system. The Gui class itself has the code to build and display the Gui. It should not also have code to implement the business logic nor to maintain the data associated with the system.

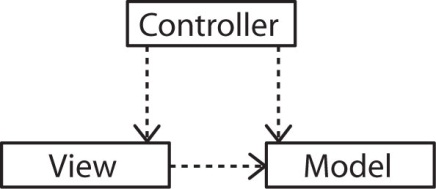
We will see examples of both of these in the detailed example that follows.

A good reference on architectural principles is: <https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles>

# MVC (Model-View-Controller) Architecture

The Model-View-Controller (MVC) is an architectural pattern that separates the modeling of the domain, the presentation, and the user actions into separate components.

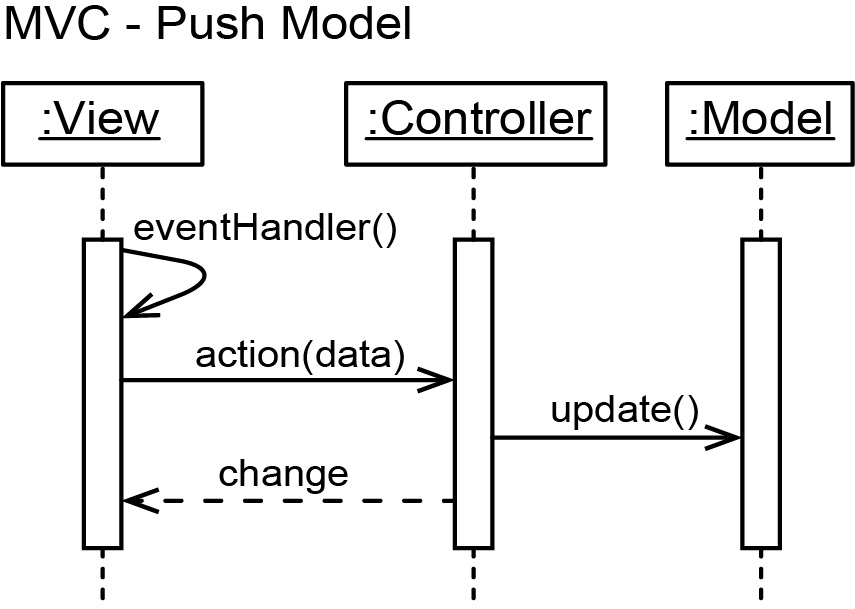
1. **Model** – The model manages the behavior and data of the application domain, responds to requests for information about its state (from the Controller or the View), and responds to instructions to change state (usually from the Controller).
2. **View** – The view manages the display of information.
3. **Controller** – The controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.

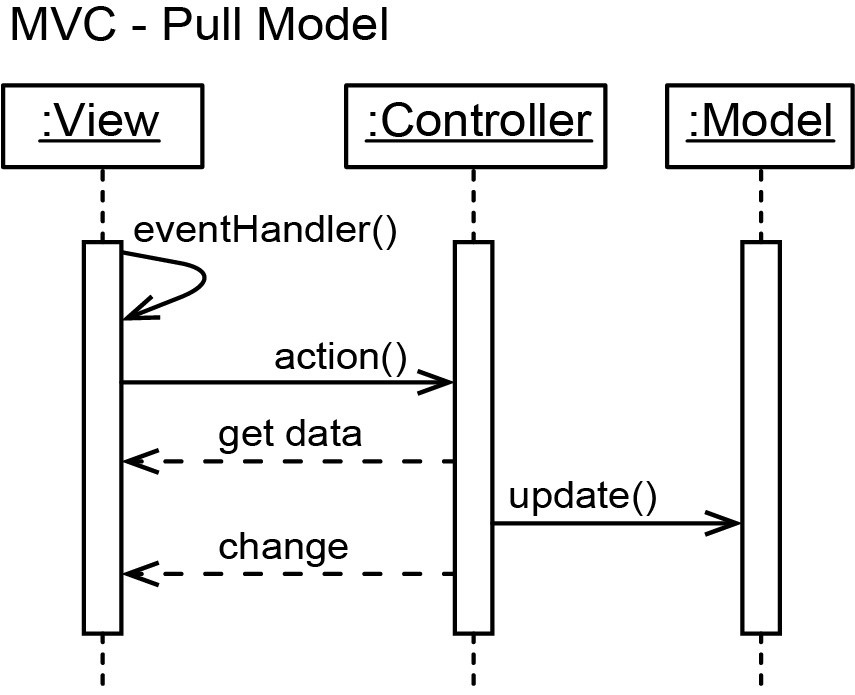
Consider the dependencies in this approach:

1. Controller depends on View and Model
2. View depends on Model
3. Model does not depend on either

Some benefits of this separation:

1. Model can be built and tested independently of the other components
2. Model can be reused
3. Alternate Views (or Controllers) can be integrated more easily.

One approach to implementing this is to use the *push* model (which is what we use in the example below). Consider the sequence diagram below. There, the user interacts with a UI, which triggers an *eventHandler*. The *eventHandler* gathers any required *data* and sends it to the *Controller* via a call to its *action* method. The *action* method may do some processing, validate data, *etc.,* and then it *updates* the *Model*. Next, the *action* method changes the *View* appropriately. It could do this directly (as shown below), or it could call a *View* method to achieve this.

Another approach is the *pull* model where no data is pushed to the *Controller*. Instead, the *Controller* gets the required data itself, either directly (as shown below) or by calling a method(s).

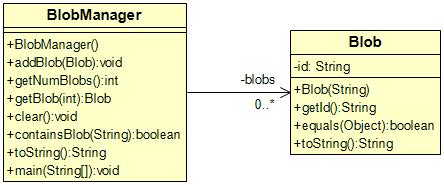
# Example

This is a very simple example that utilizes MVC. The code for this example is contained in a download on the Schedule.

## Domain Classes

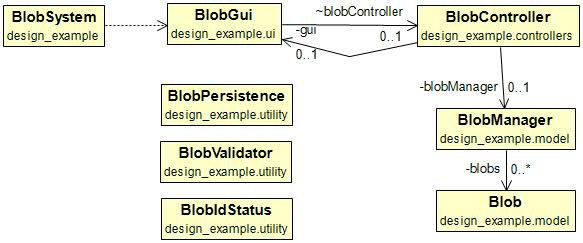
*Domain* classes are the classes that represent the naturally occurring objects in a problem. They are abstractions of the objects that must be represented in a system.

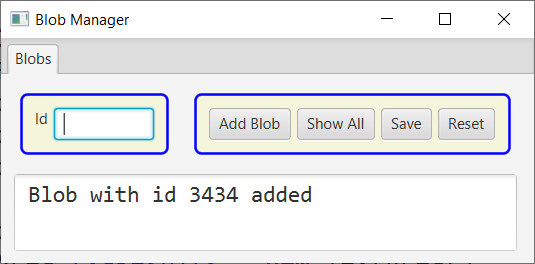
Consider the situation where we want to be able to create and manage a collection of *Blob* objects where blobs simply have an *id* which is composed of 4 digits and no two blobs can have the same *id.* The domain classes are:



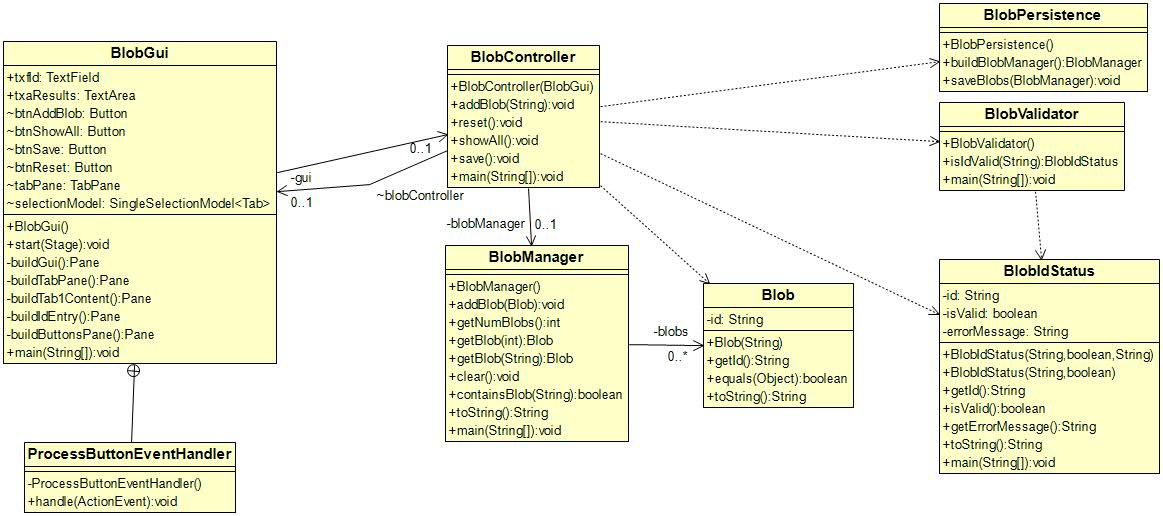
## System Classes

To build a system, we must also provide classes to support the domain classes so that they can be used to fulfill the requirements of the system. Together, with the domain classes, we call these the *system* classes. The class diagram below shows these classes for the blob system. We have organized them using the MVC architectural pattern. We discuss this in class.





The full class diagram is shown below. We will discuss this in class (or video) as well as look carefully at the actual code. The code is available on the Schedule.



## Starting the System

The *BlobGui* class creates the controller and passes a reference to itself:

**public** **class** BlobGui **extends** Application {

...

BlobController blobController;

@Override

**public** **void** start(Stage primaryStage) {

**try** {

blobController = **new** BlobController(**this**);

The *BlobController’s* constructor stores a reference to the GUI, and creates the *BlobManager* (model):

**public** **class** BlobController {

**private** BlobGui gui;

**private** BlobManager blobManager;

**public** BlobController(BlobGui gui) {

**this**.gui = gui; // Link Controller to View

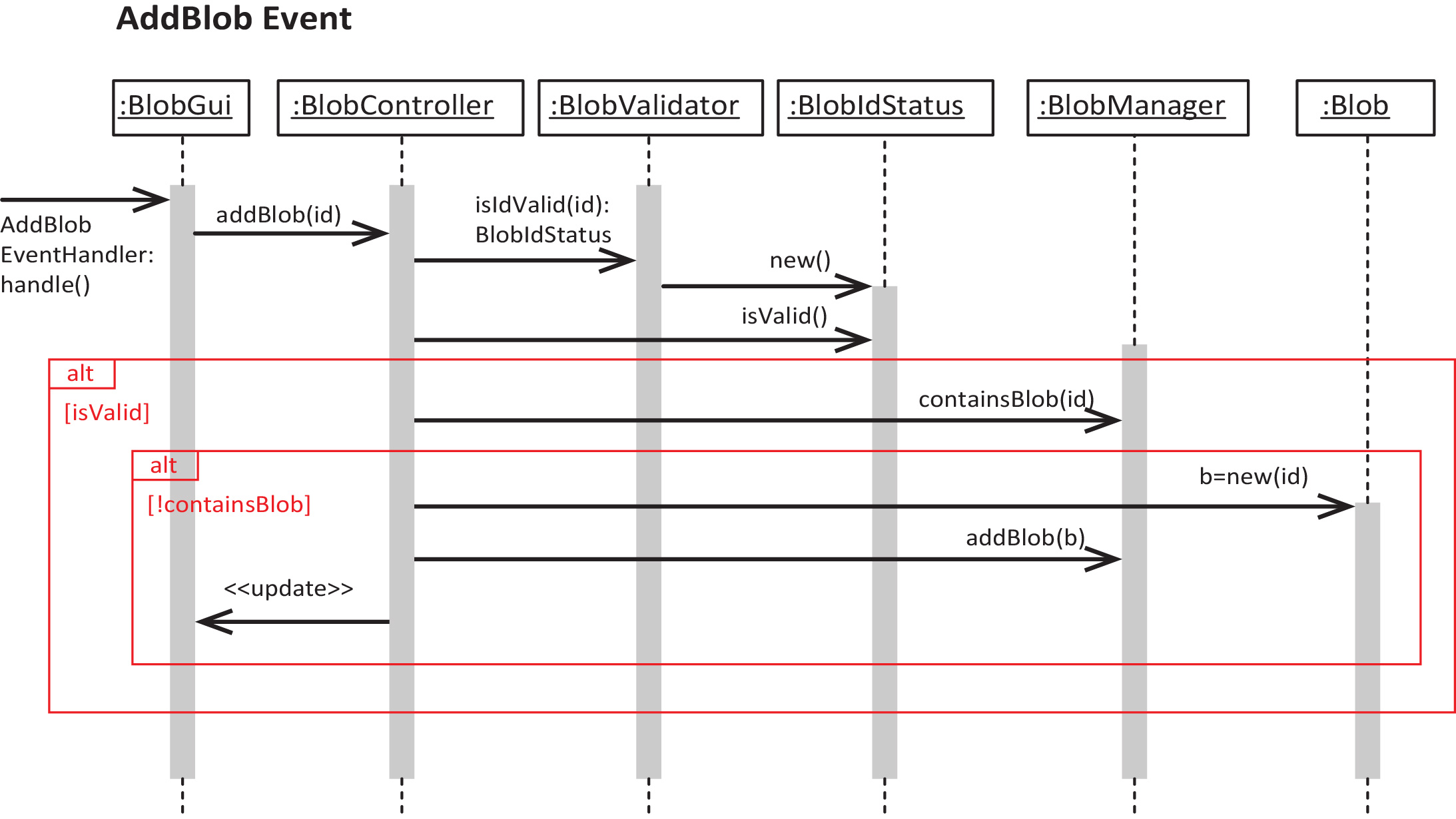
// Link Controller to Model. Reads data from disc to build Model.

**this**.blobManager = BlobPersistence.*buildBlobManager*();

}

## Use Case: Add a Blob

The sequence diagram below shows the method calls to add a *Blob* to the system:



When the user enters an Id and presses: “Add Blob”, the event handler below, gets the id from the Gui and passes it to the controller’s *addBlob* method:

**private** **class** AddBlobEventHandler **implements** EventHandler<ActionEvent> {

@Override

**public** **void** handle(ActionEvent event) {

String id = txfId.getText();

blobController.addBlob(id);

}

}

The controller’s *addBlob* method first validates the id returning a *BlobIdStatus* object. Then the *BlobIdStatus* is queried to see if the id is valid. If it is, a check is made to see if there is a *Blob* with that id that already exists. Finally, if not, a *Blob* is created and passed to the *BlobManager’s addBlob* method to be added to the *BlobManger*, and the GUI is updated.

**public** **void** addBlob(String id) {

// Check to see if id is valid

BlobIdStatus blobMessage = BlobValidator.*isIdValid*(id);

**if**(blobMessage.isValid()) { // If id is valid

**if**(!blobManager.containsBlob(id)) { // If Blob with id doesn't already exist

// Update Model - add Blob to BlobManager

Blob b = **new** Blob(id);

blobManager.addBlob(b);

// Update View - display confirmation of add

gui.txfId.setText("");

gui.txfId.requestFocus();

gui.txaResults.setText(String.*format*("Blob with id %s added", b.getId()));

}

**else** { // If Blob with id already exists

// Update View - display not added message

gui.txfId.requestFocus();

gui.txaResults.setText(String.*format*("Blob with id: %s not added\nBlob with

that id already exists", id));

}

}

**else** { // If id is not valid

// Update View - display message about why id is invalid

gui.txfId.requestFocus();

gui.txaResults.setText(String.*format*("Blob with id: %s not added\nerror message=%s",

id, blobMessage.getErrorMessage()));

}

The *BlobManager:*

**public** **class** BlobManager {

**private** Map<String, Blob> blobs = **new** TreeMap<>();

**public** BlobManager() {}

**public** **void** addBlob(Blob b) {

blobs.put(b.getId(), b);

}

## Design Principles

These are the Design Principles in [*Head First Design Patterns*](https://www.oreilly.com/library/view/head-first-design/9781492077992/).

1. Identify the aspects of your application that vary and separate them from what stays the same.
2. Program to an interface, not an implementation.
3. Favor composition over inheritance.
4. Strive for loosely coupled designs between objects that interact.
5. Classes should be open for extension, but closed for modification.
6. Dependency Inversion Principle - Depend upon abstractions. Do not depend upon concrete classes.
7. Principle of Least Knowledge – Talk only to your immediate friends
8. Hollywood Principle – Don’t call us, we’ll call you.
9. A class should have only one reason to change.
10. Resources

|  |  |
| --- | --- |
| **Site** | **Description** |
| [codementor](https://www.codementor.io/@clintwinter/notes-on-head-first-object-oriented-analysis-design-chapter-8-design-principles-197d0a5swg) | Discusses 4 design principles, with examples, towards the end. Description for SRP is good, example is not so good. |
| [Nishant Dania](https://nishantdania.com/blog/head-first-design-patterns) | Very short summary of HFDP and design principles. No examples. |
| [Adil at Work](https://adilatwork.blogspot.com/2015/11/head-first-design-patterns-by-eric.html) | Short summary of HFDP and design principles. No examples. |
| [StackExchange](https://softwareengineering.stackexchange.com/questions/337413/what-does-it-mean-when-one-says-encapsulate-what-varies) | Discussion of the *Encapsulate what varies* design principle from HFDP. Most of discussion doesn’t present in the light of using classes to encapsulate things that are subject to change. |

1. <https://www.baeldung.com/java-single-responsibility-principle> [↑](#footnote-ref-1)
2. <https://reflectoring.io/single-responsibility-principle/> [↑](#footnote-ref-2)
3. <https://www.oodesign.com/single-responsibility-principle> [↑](#footnote-ref-3)
4. <https://www.geeksforgeeks.org/single-responsibility-principle-in-java-with-examples/> [↑](#footnote-ref-4)