Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Basics of Mobile Application Development

Design patterns

Design patterns

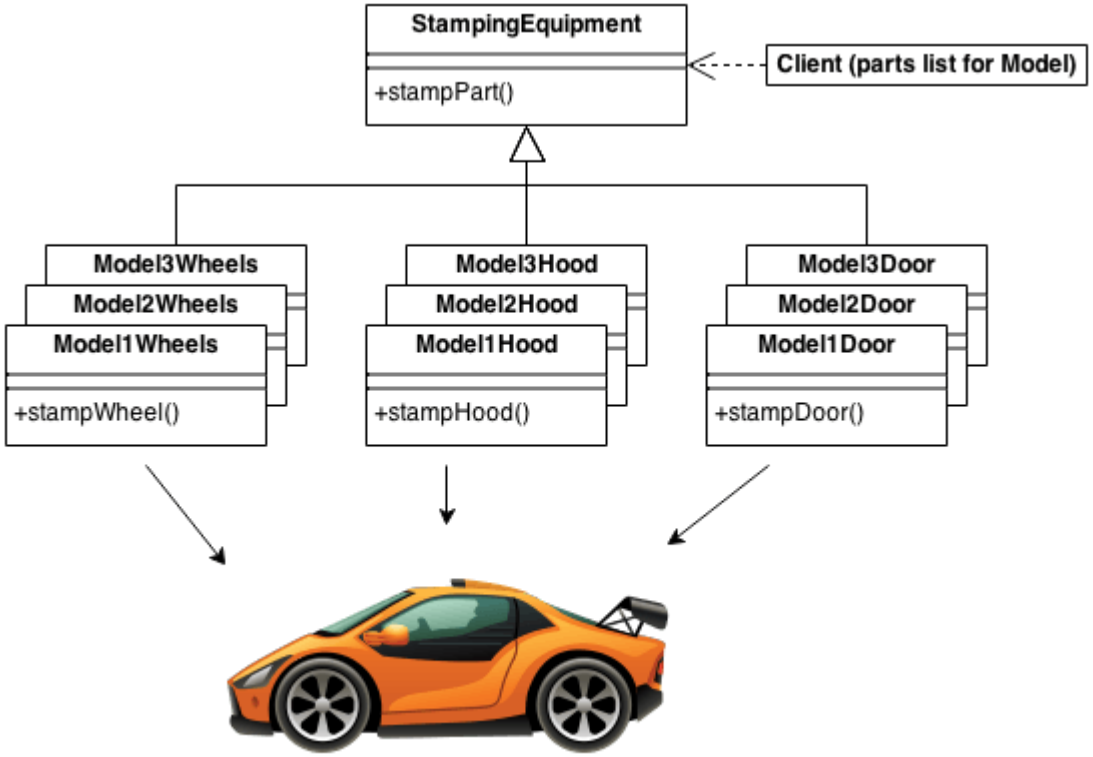
- Design patterns are description of collaborative objects and classes.
- General patterns can be specialized and implemented in order to solve software design issues.
  - It is beneficial to use object-oriented programming languages (C++, Java, Smalltalk, …)
  - In case of procedural languages further patterns may be required:
    - Inheritance
    - Encapsulation
    - Polymorphism
- Categories
  - Objective
    - Creational patterns
    - Structural patterns
    - Behavioral patterns
  - Scope
    - Class
    - Object

Some of patterns

- Creational
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral
  - Chain of Responsibility
  - *Command*
  - Interpreter
  - *Iterator*
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Abstract Factory

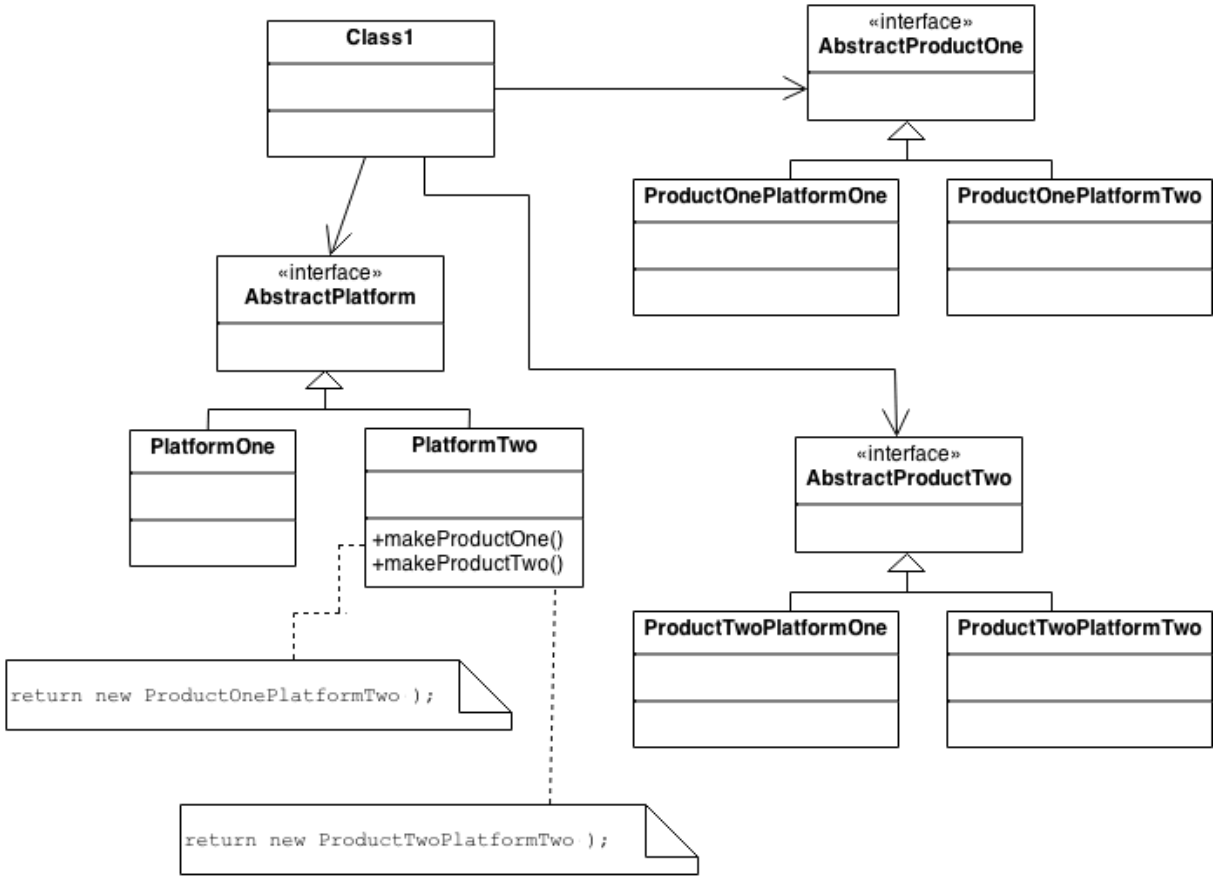
- Creational Pattern
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".



Abstract Factory

- Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes.
  - The "factory" object has the responsibility for providing creation services for the entire platform family.
  - Clients never create platform objects directly, they ask the factory to do that for them.

Abstract Factory



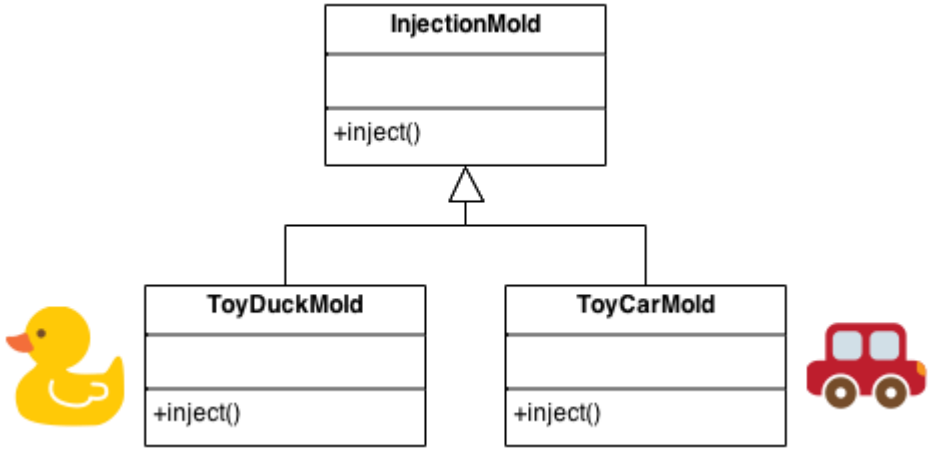
Abstract Factory

```
abstract class CPU {}

class EmberCPU extends CPU {}

class EnginolaCPU extends CPU {}

abstract class MMU {}

class EmberMMU extends MMU {}

class EnginolaMMU extends MMU {}

class EmberToolkit extends AbstractFactory {
    @Override
    public CPU createCPU() {
        return new EmberCPU();
    }

    @Override
    public MMU createMMU() {
        return new EmberMMU();
    }
}

class EnginolaToolkit extends AbstractFactory {
    @Override
    public CPU createCPU() {
        return new EnginolaCPU();
    }

    @Override
    public MMU createMMU() {
        return new EnginolaMMU();
    }
}

enum Architecture {
    ENGINOLA, EMBER
}
```

```
abstract class AbstractFactory {
    private static final EmberToolkit EMBER_TOOLKIT = new
EmberToolkit();
    private static final EnginolaToolkit ENGINOLA_TOOLKIT = new
EnginolaToolkit();

static AbstractFactory getFactory(Architecture architecture) {
        AbstractFactory factory = null;
        switch (architecture) {
            case ENGINOLA:
                factory = ENGINOLA_TOOLKIT;
                break;
            case EMBER:
                factory = EMBER_TOOLKIT;
                break;
        }
        return factory;
    }

    public abstract CPU createCPU();

    public abstract MMU createMMU();
}

public class Client {
    public static void main(String[] args) {
        AbstractFactory factory =
AbstractFactory.getFactory(Architecture.EMBER);
        CPU cpu = factory.createCPU();
    }
}
```

Factory Method

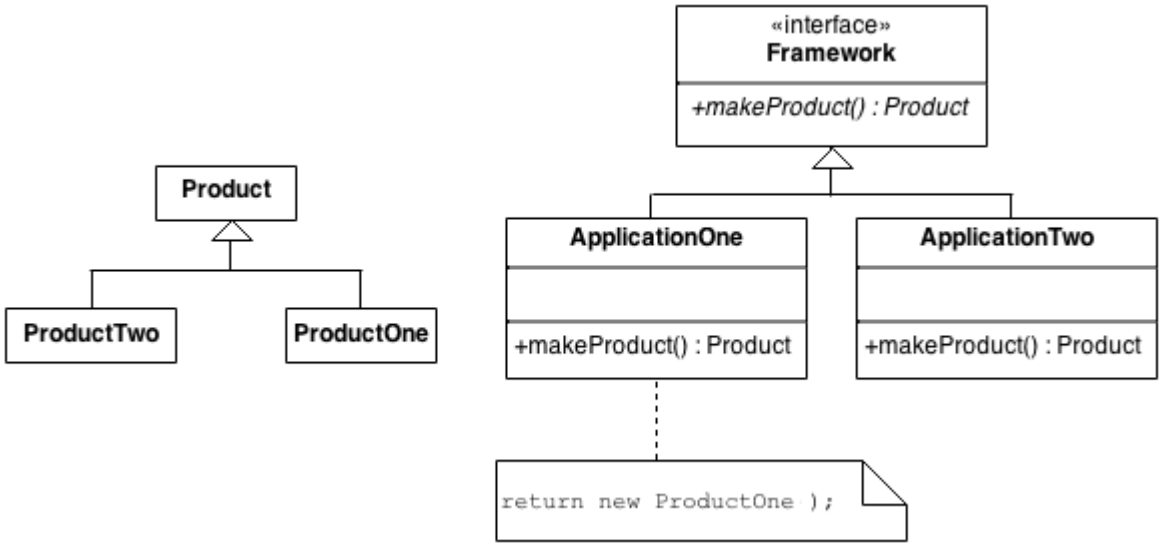
- Creational pattern
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
  - Defining a "virtual" constructor.



Factory Method

- A framework needs to standardize the architectural model for a range of applications
  - But allow for individual applications to define their own domain objects and provide for their instantiation.
- Factory Method is to creating objects as Template Method is to implementing an algorithm.
  - A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps)
  - Then delegates the creation details to subclasses that are supplied by the client.

Factory Method



Factory Method

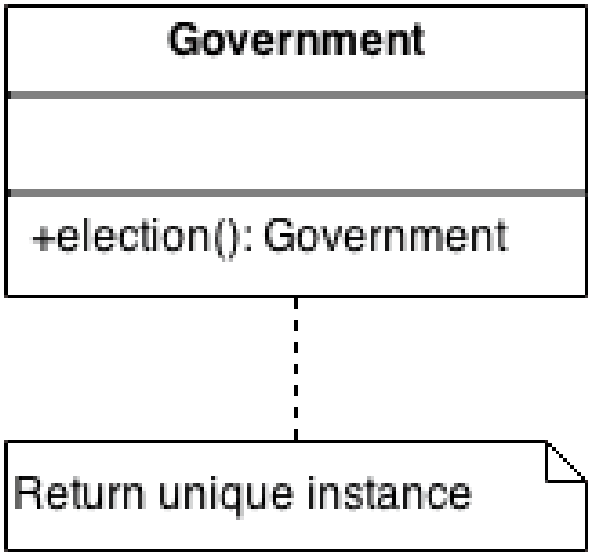
```
interface ImageReader {
    DecodedImage getDecodeImage();
}

class DecodedImage {
    private String image;

    public DecodedImage(String image) {
        this.image = image;
    }

    @Override
    public String toString() {
        return image + ": is decoded";
    }
}

class GifReader implements ImageReader {
    private DecodedImage decodedImage;

    public GifReader(String image) {
        this.decodedImage = new DecodedImage(image);
    }

    @Override
    public DecodedImage getDecodeImage() {
        return decodedImage;
    }
}
```

```
class JpegReader implements ImageReader {
    private DecodedImage decodedImage;

    public JpegReader(String image) {
        decodedImage = new DecodedImage(image);
    }

    @Override
    public DecodedImage getDecodeImage() {
        return decodedImage;
    }
}

public class FactoryMethodDemo {
    public static void main(String[] args) {
        DecodedImage decodedImage;
        ImageReader reader = null;
        String image = args[0];
        String format = image.substring(image.indexOf('.') + 1,
(image.length()));
        if (format.equals("gif")) {
            reader = new GifReader(image);
        }
        if (format.equals("jpeg")) {
            reader = new JpegReader(image);
        }
        assert reader != null;
        decodedImage = reader.getDecodeImage();
        System.out.println(decodedImage);
    }
}
```

Singleton

- Creational pattern
- Prohibits to create more than one instance of a class
  - Allows to create one and provides an access point to it
- Example
  - Only a single file system manager, or window manager is allowed
- Implementation
  - A hidden class function can only instantiate the class, ensuring that only one instance is allowed
  - Subclasses may be created

Singleton



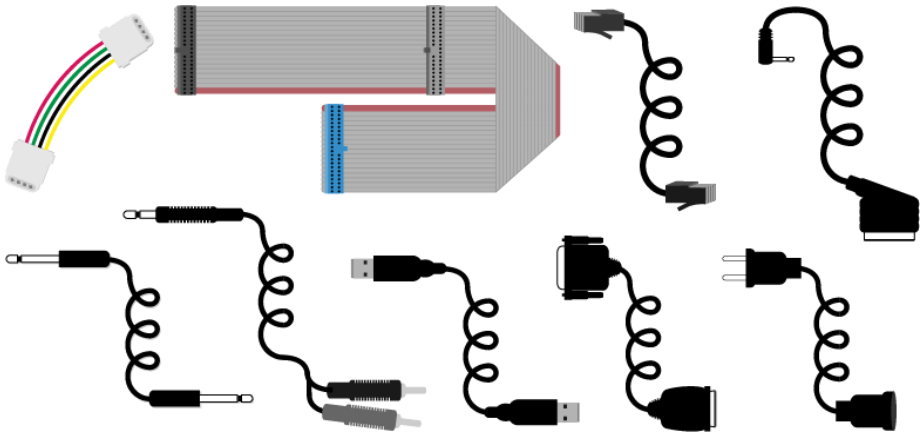
Singleton

```
#include <iostream>
#include <string>
#include <stdlib.h>
using namespace std;

class Number
{
  public:
    // 2. Define a public static accessor func
    static Number *instance();
    static void setType(string t)
    {
      type = t;
      delete inst;
      inst = 0;
    }
    virtual void setValue(int in)
    {
      value = in;
    }
    virtual int getValue()
    {
      return value;
    }
  protected:
    int value;
    // 4. Define all ctors to be protected
    Number()
    {
        cout << ":ctor: ";
    }
  // 1. Define a private static attribute
  private:
    static string type;
    static Number *inst;
};

string Number::type = "decimal";
Number *Number::inst = 0;
```

```
class Octal: public Number
{
  // 6. Inheritance can be supported
  public:
    friend class Number;
    void setValue(int in)
    {
      char buf[10];
      sprintf(buf, "%o", in);
      sscanf(buf, "%d", &value);
    }
  protected:
    Octal(){}
};

Number *Number::instance()
{
  if (!inst)
    // 3. Do "lazy initialization" in the accessor function
    if (type == "octal")
      inst = new Octal();
    else
      inst = new Number();
  return inst;
}

int main()
{
  // Number  myInstance; - error: cannot access protected constructor
  // 5. Clients may only use the accessor function to manipulate the
Singleton
  Number::instance()->setValue(42);
  cout << "value is " << Number::instance()->getValue() << endl;
  Number::setType("octal");
  Number::instance()->setValue(64);
  cout << "value is " << Number::instance()->getValue() << endl;
}
```

Adapter

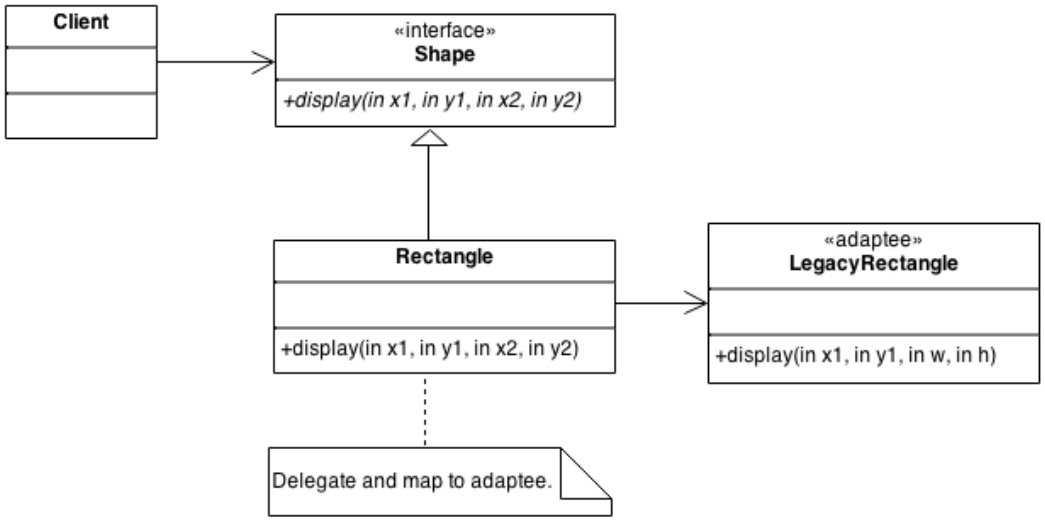
- Structural class/object pattern
  - An "off the shelf" component offers compelling functionality that you would like to reuse
  - But its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

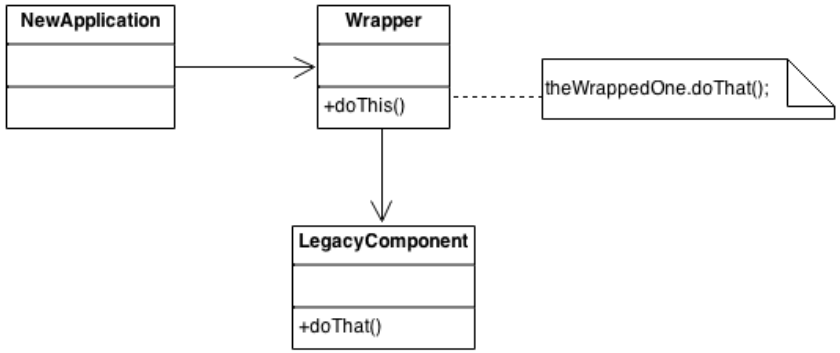


Adapter

- Reuse has always been painful and elusive.
  - One reason has been the tribulation of designing something new, while reusing something old.
  - There is always something not quite right between the old and the new.
  - It may be physical dimensions or misalignment. It may be timing or synchronization.
  - It may be unfortunate assumptions or competing standards.

Adapter


Client
«interface»
Shape
+display(in x1, in y1, in x2, in y2)
Rectangle
+display(in x1, in y1, in x2, in y2)
«adaptee»
LegacyRectangle
+display(in x1, in y1, in w, in h)
Delegate and map to adaptee.


NewApplication
Wrapper
+doThis()
theWrappedOne.doThat();
LegacyComponent
+doThat()

Adapter

```
import abc

class Target(metaclass=abc.ABCMeta):
    """
    Define the domain-specific interface that Client uses.
    """

    def __init__(self):
        self._adaptee = Adaptee()

    @abc.abstractmethod
    def request(self):
        pass


class Adapter(Target):
    """
    Adapt the interface of Adaptee to the Target
interface.
    """

    def request(self):
        self._adaptee.specific_request()
```

```
class Adaptee:
    """
    Define an existing interface that needs adapting.
    """

    def specific_request(self):
        pass


def main():
    adapter = Adapter()
    adapter.request()


if __name__ == "__main__":
    main()
```

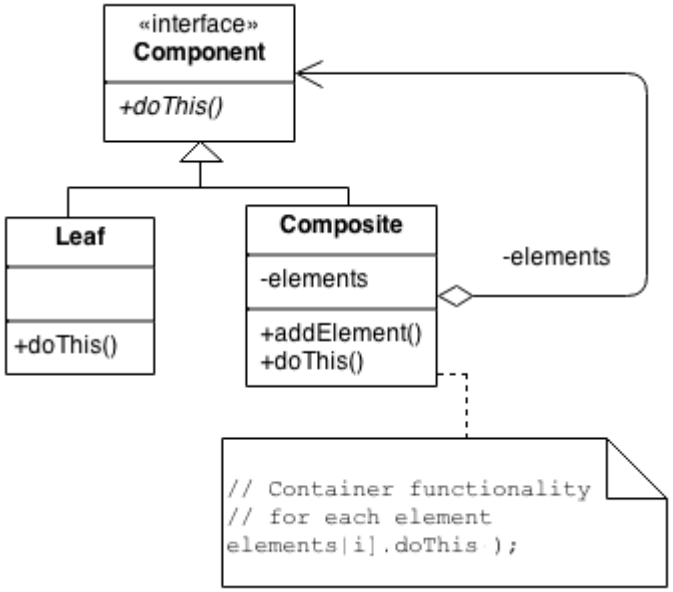
Composition

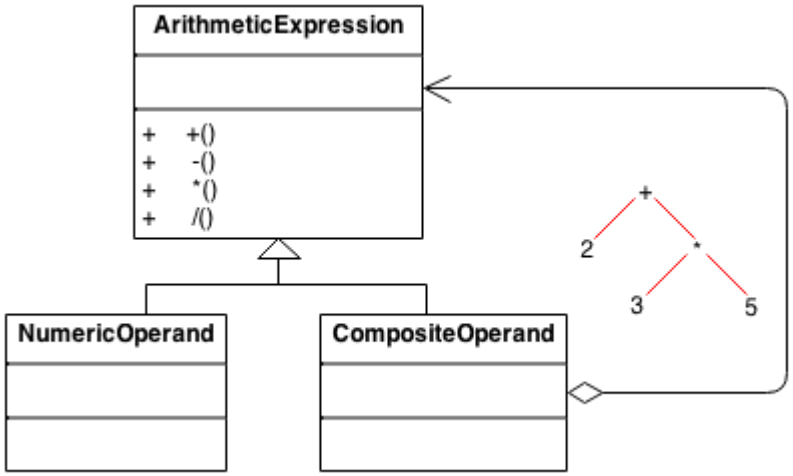
- Structural pattern
  - Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects.
    - Processing of a primitive object is handled one way, and processing of a composite object is handled differently.
    - Having to query the "type" of each object before attempting to process it is not desirable.
  - For example in a graphical application, the basic and complex view components can be treated with the same functions
    - Composite elements can be created by grouping objects

Composition

- Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects.
  - Subclass the Primitive and Composite classes off of the Component class.
  - Each Composite object "couples" itself only to the abstract type Component as it manages its "children".
- Use this pattern whenever you have "composites that contain components, each of which could be a composite".

Composition



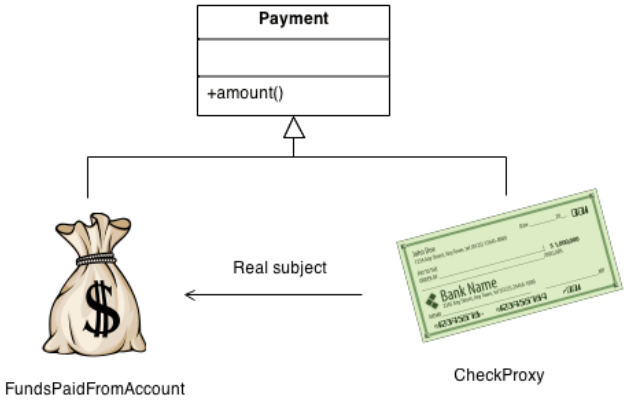


```
#include <iostream>
#include <vector>
using namespace std;

class Component
{
  public:
    virtual void traverse() = 0;
};

class Primitive: public Component
{
    int value;
  public:
    Primitive(int val)
    {
        value = val;
    }
    void traverse()
    {
        cout << value << "   ";
    }
};

class Composite: public Component
{
    vector < Component * > children;
    int value;
  public:
    Composite(int val)
    {
        value = val;
    }
    void add(Component *c)
    {
        children.push_back(c);
    }
    void traverse()
    {
        cout << value << "   ";
        for (int i = 0; i < children.size(); i++)
          children[i]->traverse();
    }
};
```

```
class Row: public Composite
{
  public:
    // Two different kinds of "con-
    Row(int val): Composite(val){}
    // tainer" classes.  Most of the
    void traverse()
    {
        // "meat" is in the Composite
        cout << "Row"; // base class.
        Composite::traverse();
    }
};

class Column: public Composite
{
  public:
    Column(int val): Composite(val){}
    void traverse()
    {
        cout << "Col";
        Composite::traverse();
    }
};

int main()
{
  Row first(1);                  // Row1
  Column second(2);              //   |
  Column third(3);               //   +-- Col2
  Row fourth(4);                 //   |     |
  Row fifth(5);                  //   |     +-- 7
  first.add(&second);            //   +-- Col3
  first.add(&third);             //   |     |
  third.add(&fourth);            //   |     +-- Row4
  third.add(&fifth);             //   |     |     |
  first.add(&Primitive(6));      //   |     |     +-- 9
  second.add(&Primitive(7));     //   |     +-- Row5
  third.add(&Primitive(8));      //   |     |     |
  fourth.add(&Primitive(9));     //   |     |     +-- 10
  fifth.add(&Primitive(10));     //   |     +-- 8
  first.traverse();              //   +-- 6
  cout << '\n';
}
```

Proxy

- Structural pattern
- To control an object though another object
- You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.



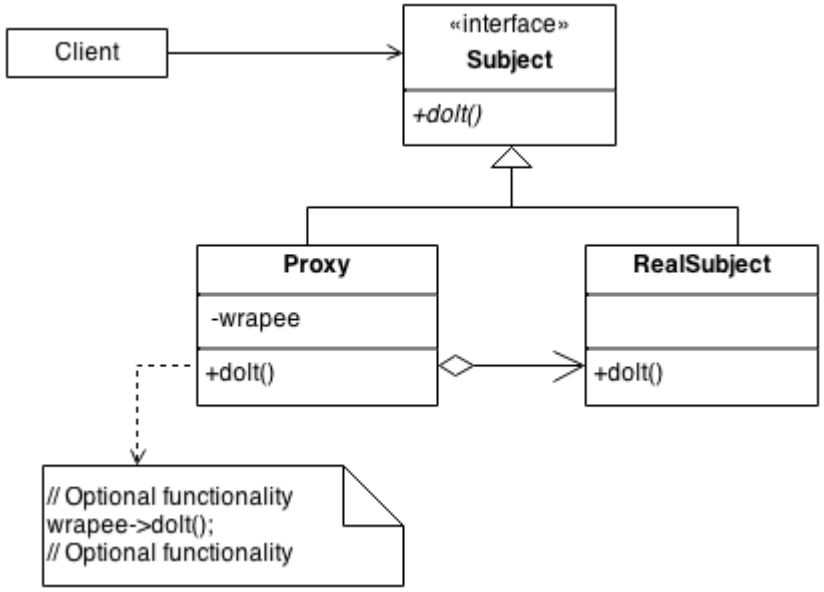
Proxy

- Design a surrogate, or proxy, object that:
  - instantiates the real object the first time the client makes a request of the proxy
  - remembers the identity of this real object
  - forwards the instigating request to this real object
- Then all subsequent requests are simply forwarded directly to the encapsulated real object.

Proxy

- There are four common situations in which the Proxy pattern is applicable.
  - A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
  - A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
  - A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
  - A smart proxy interposes additional actions when an object is accessed. Typical uses include:
    - Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
    - Loading a persistent object into memory when it's first referenced,
    - Checking that the real object is locked before it is accessed to ensure that no other object can change it.

Proxy


Client
«interface»
Subject
+dolt()
Proxy
-wrapee
+dolt()
RealSubject
+dolt()
// Optional functionality
wrapee->dolt();
// Optional functionality

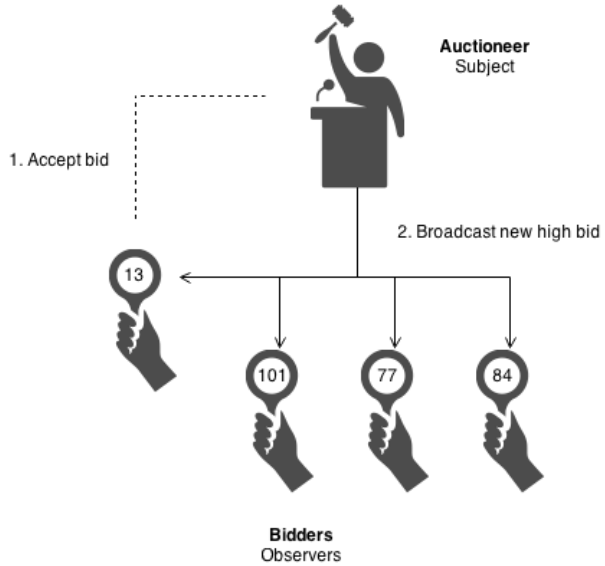
Proxy

```
interface SocketInterface {
    String readLine();
    void  writeLine(String str);
    void  dispose();
}

class SocketProxy implements SocketInterface {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;

    public SocketProxy(String host, int port, boolean wait) {
        try {
            if (wait) {
                // 2. Encapsulate the complexity/overhead of the target in the wrapper
                ServerSocket server = new ServerSocket(port);
                socket = server.accept();
            } else {
                socket = new Socket(host, port);
            }
            in  = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }

    public String readLine() {
        String str = null;
        try {
            str = in.readLine();
        } catch( IOException e ) {
            e.printStackTrace();
        }
        return str;
    }
```

```
    public void writeLine(String str) {
        // 4. The wrapper delegates to the target
        out.println(str);
    }

    public void dispose() {
        try {
            socket.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}

public class ProxyDemo {
    public static void main( String[] args ) {
        // 3. The client deals with the wrapper
        SocketInterface socket = new SocketProxy( "127.0.0.1", 8080, args[0].equals("first") ? true : false );
        String  str;
        boolean skip = true;
        while (true) {
            if (args[0].equals("second") && skip) {
                skip = !skip;
            } else {
                str = socket.readLine();
                System.out.println("Receive - " + str);
                if (str.equals(null)) {
                    break;
                }
            }
            System.out.print( "Send ---- " );
            str = new Scanner(System.in).nextLine();
            socket.writeLine( str );
            if (str.equals("quit")) {
                break;
            }
        }
        socket.dispose();
    }
}
```

Observer

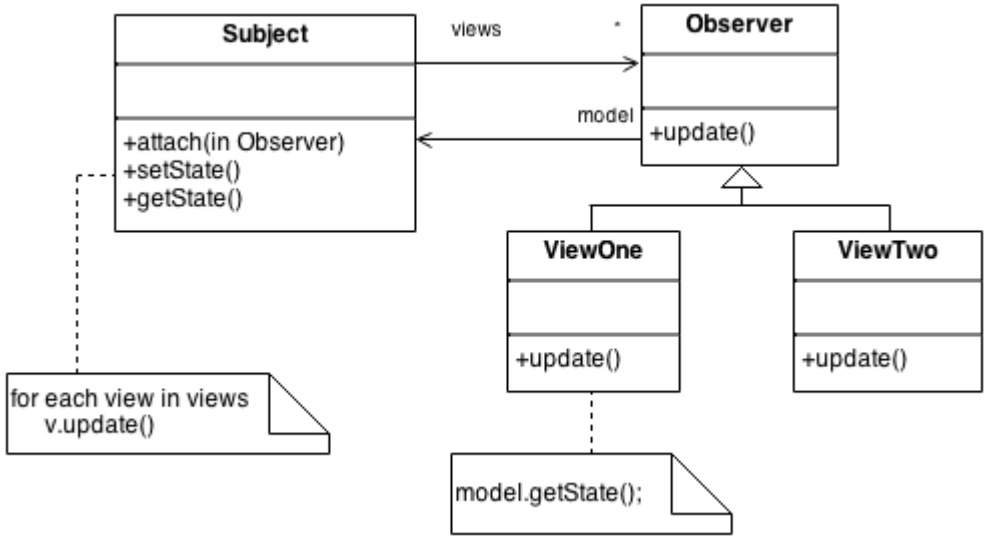
- Behavioral pattern
- Creates a 1 to N dependency between objects
  - When the state of the observed object changes the others are notified about the event and also can change their state



Observer

- Define an object that is the "keeper" of the data model and/or business logic (the Subject).
  - Delegate all "view" functionality to decoupled and distinct Observer objects.
  - Observers register themselves with the Subject as they are created.
  - Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.
- This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.

Observer


Subject
views
Observer
model
+attach(in Observer)
+setState()
+getState()
+update()
ViewOne
ViewTwo
+update()
+update()
for each view in views
v.update()
model.getState();

Observer

```
interface AlarmListener {
    void alarm();
}

class SensorSystem {
    private Vector listeners = new Vector();

    public void register(AlarmListener alarmListener) {
        listeners.addElement(alarmListener);
    }

    public void soundTheAlarm() {
        for (Enumeration e = listeners.elements();
e.hasMoreElements();) {
            ((AlarmListener) e.nextElement()).alarm();
        }
    }
}

class Lighting implements AlarmListener {
    public void alarm() {
        System.out.println("lights up");
    }
}

class Gates implements AlarmListener {
    public void alarm() {
        System.out.println("gates close");
    }
}
```

```
class CheckList {
    // Template Method design pattern
    public void byTheNumbers() {
        localize();
        isolate();
        identify();
    }

    protected void localize() {
        System.out.println("   establish a perimeter");
    }

    protected void isolate() {
        System.out.println("   isolate the grid");
    }

    protected void identify() {
        System.out.println("   identify the source");
    }
}

// class inherit.
// type inheritance
class Surveillance extends CheckList implements AlarmListener {
    public void alarm() {
        System.out.println("Surveillance - by the numbers:");
        byTheNumbers();
    }

    protected void isolate() {
        System.out.println("   train the cameras");
    }
}
```

Delegate

- A specific object do not execute its task, instead of it delegates the task to another object
  - The other can be considered as a helper-object
- The responsibility is also delegated
  - It considered as a server, with responsibility
- Implementation
  - Through interface classes
  - In case of a function call, the implementation refers to an implementation of other class

Delegation – Example

```
class I {
public:
        virtual void f() = 0;
        virtual void g() = 0;
        virtual ~I() {}
};
class A : public I {
public:
        void f() { cout << "A: doing f()" << endl; }
        void g() { cout << "A: doing g()" << endl; }
        ~A() { cout << "A: cleaning up." << endl; }
};
class B : public I {
public:
        void f() { cout << "B: doing f()" << endl; }
        void g() { cout << "B: doing g()" << endl; }
        ~B() { cout << "B: cleaning up." << endl; }
};
```

Delegation – Example

```
class C : public I {
  public:
    C() : i( new A() ) { }
    virtual ~C() { delete i; }
  private:
    I* i;
  public:
    void f() { i->f(); }
    void g() { i->g(); }
    void toA() { delete i; i = new A(); }
    void toB() { delete i; i = new B(); }
};

int main() {
    C c;
    c.f();   //A: doing f()
    c.g();   //A: doing g()
    c.toB(); //A: cleaning up.
    c.f();   //B: doing f()
    c.g();   //B: doing g()
}
```

Target-Action

- Used in event-driven applications
  - Programs with GUI
- The objects of the program are in dynamic connection with each other
  - Target: a specific object that is the target of a message (task)
  - Action: what target should execute

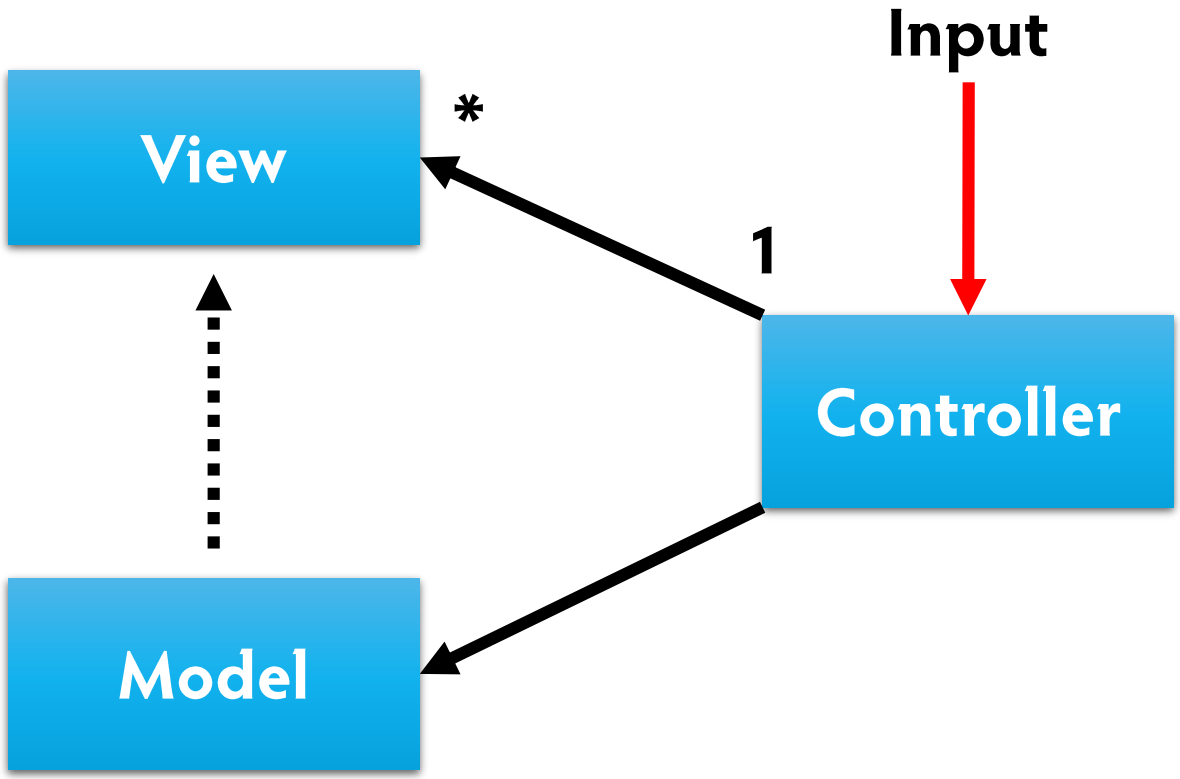
`ObjectiveC` example

```
[button setTarget: self];
[button setAction: @selector(doSomething)];
```

Design paradigms

- Complex structural patterns
- The design of entire structure is governed by these principles
  - Modell-View-Controller
  - Modell-View-Presenter
  - Modell-View-ViewModell
  - Target-Action

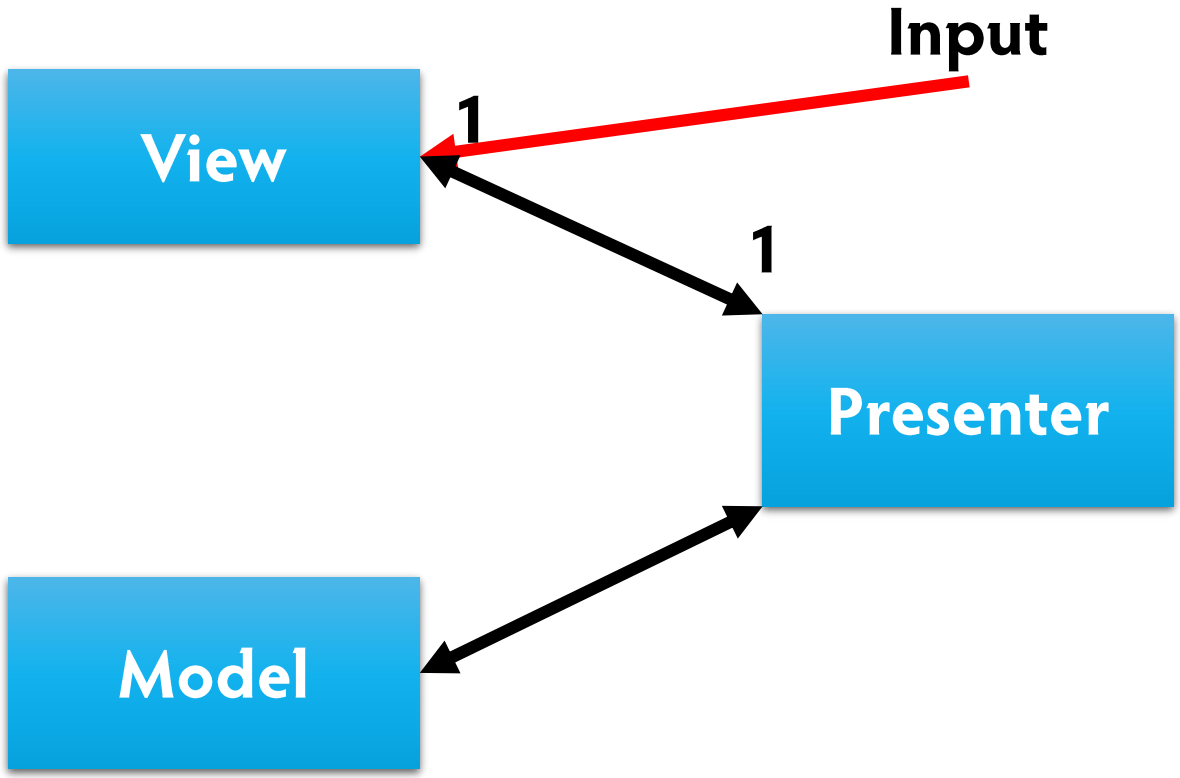
MVC



MVC

- The application has three layers
  - Model
    - The representation of the information stored by the application
      - Plain data is augmented with meta data to provide meaning
    - Applications use permanent storing procedures to save data
    - The data access layer is part of the model, most of the cases
  - View
    - Visualize the model in the correct form, which is capable of user interaction
    - Typically it is a UI element
    - Different view for different objective may be exist
  - Controller
    - Events (mostly user interactions) are processed and appropriate response is generated
    - May change the model

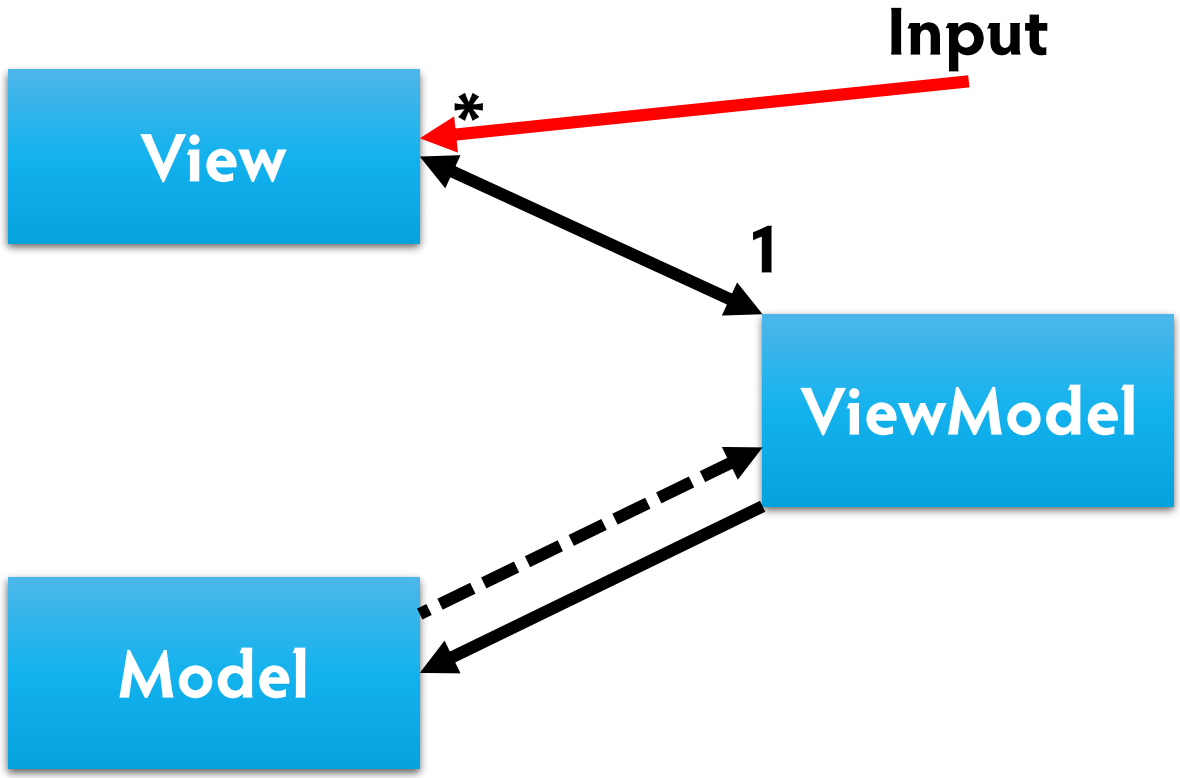
MVP



MVP

- The application has three layers
  - Model
    - The representation of the information stored by the application
      - Plain data is augmented with meta data to provide meaning
    - Applications use permanent storing procedures to save data
    - The data access layer is part of the model, most of the cases
  - View
    - Visualize the model in the correct form, which is capable of user interaction
    - Accepts the user interaction on View and send to the Presenter
      - Passive View: only displays the data, all business logic and transformation is in the Presenter
      - Supervisor View: some of the control tasks are here. It simplifies the Presenter by removing the conversions, checks, UI repaint/redraw procedures. Thus in Presenter only the business logic should be implemented.
  - Presenter
    - Middle layer, keeps the application in one piece
    - Business logic, and process control
    - Transforms data and transport between View and Model layer

MVVM



MVVM

- This pattern is similar to the MVC pattern
  - Designed for event-driven applications
  - Model
    - The representation of the information stored by the application
      - Plain data is augmented with meta data to provide meaning
    - Application use permanent storing procedures to save data
    - The data access layer is part of the model, most of the cases
  - View
    - Visualize the model in the correct form, which is capable of user interaction
    - Typically it is a UI element
    - Different view for different objective may be exist
  - ViewModel
    - Model of View
    - It can be considered as a special Controller, which converts the information coming from the Model to View, and the commands coming from the View to the Model
    - Public properties, commands and abstract interface are provided
    - It represents the conceptional state of data, instead of the real data which is stored in the model

Tests, software design

Software design

- Remember / recall
  - Basics of Software Technology
- Most important
  - Reasoned plan and software
  - Even drawings, diagrams, etc.
- Debugging is not trivial
  - But not impossible as well
- Emulator / simulator is not perfect
  - Some of the functions are missing
  - Varying quality of hardware and implementation
    - The specifications are not followed precisely

Test – Validation

- Verification and validation – objectives:
  - To find the errors in the system
  - To ascertain about that the system can be used in real situations
- Most widespread validation technique
- The errors themselves have to be found, not their absence
- A test is successful when at least one error is discovered
- Testing defects:
  - The objective is to discover faults and defects of the system
  - Types:
    - Component tests: black box, equivalence-classes, structural tests, path-test
    - Integration tests: „top-down/bottom-up", interface tests, stress-test
    - Object oriented tests
- Statistical tests
  - Testing the performance and reliability, in real situations (with real user input, and frequency)

Test types

- Functional tests
  - The program is considered as a black box, the test cases are made upon specification
  - The implementation is not taken into account
  - Tests can be designed in early stage if the development
  - Special knowledge may be required to design some of the tests
- Structural test
  - Tests are designed based on the structure and implementation of the program
  - Equivalence classes can be designed based on the structure and code
  - During test design the source code is analyzed to ensure that all statements are executed at least once
    - All execution path cannot be tested in reality

Test types

- OOP test
  - Component and integration test can be used in OOP systems
  - Differences
    - The object as components are larger than simple functions
      - White box test can be applied with difficulties
    - Objects are loosely coupled, and the system/subsystem may not have unambigous bottom/top
    - The source of reused components may be inaccessible this we cannot analyze
- Inspection
  - The objective of software inspection is to find defects
  - Approximately 60% of the errors can be found with the cheaper inspection
  - A single test is capable of revealing a single error, inspection can find multiple errors
  - Inspection and testing cannot substitute each other
  - Inspection should be executed in the early stages of the development, to reduce the cost of tests

Load tests

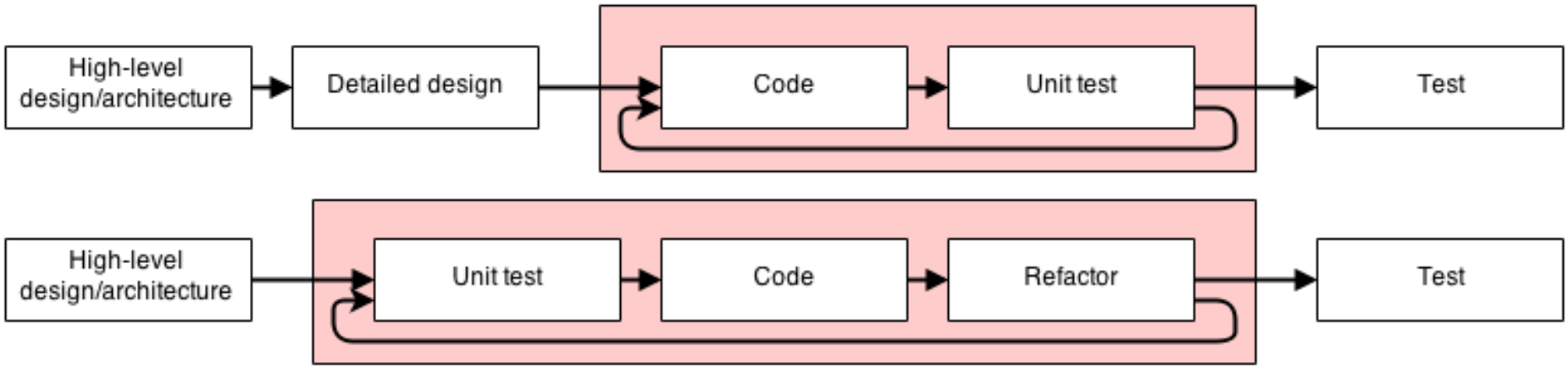
- The systems should be tested with larger load (than designed)
  - The load should be increased gradually, until system failure or performance degradation
- Tasks
  - To test the behavior of the system under extreme conditions
  - Overload must not cause data loss, or complete service failure
  - Such defects can be discovered, which do not happen under normal conditions.
- It is especially important in case of distributed systems
  - Larger load may cause coordination / load distribution problems
  - Thus it may result in increasing and self sustaining overload process

Test design

- For each program unit tests should be designed
  - Unit tests
  - Component integration tests
- Tests have to be started to design in parallel with software design process

Software life cycle– TDD

- Agile method very popular



Extreme Programming tests

- In the Extreme Programming approach,
  - Tests are written before the code itself
  - If code has no automated test case, it is assumed not to work
  - A test framework is used so that automated testing can be done after every small change to the code
    - This may be as often as every 5 or 10 minutes
  - If a bug is found after development, a test is created to keep the bug from coming back
- Consequences
  - Fewer bugs
  - More maintainable code
  - Continuous integration
    - During development, the program always works
    - it may not do everything required, but what it does, it does right

Two examples

- ```
  int max(int a, int b) {
      if (a > b) {
          return a;
      } else {
          return b;
      }
  }
  ```
- ```
  void testMax() {
      int x = max(3, 7);
      if (x != 7) {
          System.out.println("max(3, 7) gives " + x);
      }
      x = max(3, -7);
      if (x != 3) {    System.out.println("max(3, -7)
  gives " + x);
      }
  }
  ```
- ```
  public static void main(String[] args) {
      new MyClass().testMax();
  }
  ```

- ```
  @Test
  void testMax() {
      assertEquals(7, max(3, 7));
      assertEquals(3, max(3, -7));
  }
  ```

JUnit

- JUnit is a framework for writing tests
  - JUnit was written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of XP methodology)
  - JUnit uses Java's reflection capabilities (Java programs can examine their own code)
  - JUnit helps the programmer:
    - define and execute tests and test suites
    - formalize requirements and clarify architecture
    - write and debug code
    - integrate code and always be ready to release a working version
  - JUnit is not included in SDK, but almost all IDEs include it
    - You have to import it to your project

Terminology

- A test fixture sets up the data (both objects and primitives) that are needed to run tests
  - Example: If you are testing code that updates an employee record, you need an employee record to test it on
- A unit test is a test of a single class
- A test case tests the response of a single method to a particular set of inputs
- A test suite is a collection of test cases
- A test runner is software that runs tests and reports results

- An integration test is a test of how well classes work together
  - JUnit provides some limited support for integration tests

Test suite

- Obviously you have to test your code to get it working in the first place
  - You can do ad hoc testing (testing whatever occurs to you at the moment), or
  - You can build a test suite (a thorough set of tests that can be run at any time)
- Disadvantages of writing a test suite
  - It's a lot of extra programming
    - True—but use of a good test framework can help quite a bit
  - You don't have time to do all that extra work
    - False—Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite
- Advantages of having a test suite
  - Your program will have many fewer bugs
  - It will be a lot easier to maintain and modify your program
    - This is a huge win for programs that, unlike class assignments, get actual use!

Assert methods

- Each assert method has parameters like these: message, expected-value, actual-value
- Assert methods dealing with floating point numbers get an additional argument, a tolerance
- Each assert method has an equivalent version that does not take a message – however, this use is not recommended because:
  - messages helps documents the tests
  - messages provide additional information when reading failure logs

More in test classes

- Suppose you want to test a class Counter
- public class CounterTest
  extends junit.framework.TestCase {
  - This is the unit test for the Counter class
- public CounterTest() { } //Default constructor
- protected void setUp()
  - Test fixture creates and initializes instance variables, etc.
- protected void tearDown()
  - Releases any system resources used by the test fixture
- public void testIncrement(), public void testDecrement()
  - These methods contain tests for the Counter methods increment(), decrement(), etc.
  - Note capitalization convention

JUnit tests for Counter

- public class CounterTest extends junit.framework.TestCase {
  Counter counter1;
  public CounterTest() { } // default constructor
- protected void setUp() { // creates a (simple) test fixture
  counter1 = new Counter();
  }
- public void testIncrement() {
  assertTrue(counter1.increment() == 1);
  assertTrue(counter1.increment() == 2);
  }
- public void testDecrement() {
  assertTrue(counter1.decrement() == -1);
  }
  }

TestSuites

- TestSuites collect a selection of tests to run them as a unit
- Collections automatically use TestSuites, however to specify the order in which tests are run, write your own:

```
public static Test suite() {
        suite.addTest(new TestBowl("testBowl"));
        suite.addTest(new TestBowl("testAdding"));
        return suite;
}
```

- Should seldom have to write your own TestSuites as each method in your TestCase should be independent of all others
- Can create TestSuites that test a whole package:

```
public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(TestBowl.class);
        suite.addTestSuite(TestFruit.class);
        return suite;
}
```

A simple example

- Suppose that you have a class Arithmetic with methods `int multiply(int x, int y)`, and `boolean isPositive(int x)`
-
```
import org.junit.*;
import static org.junit.Assert.*;
public class ArithmeticTest {

@Test
public void testMultiply() {
    assertEquals(4, Arithmetic.multiply(2, 2));
    assertEquals(-15, Arithmetic.multiply(3, -5));
}

@Test
public void testIsPositive() {
    assertTrue(Arithmetic.isPositive(5));
    assertFalse(Arithmetic.isPositive(-5));
    assertFalse(Arithmetic.isPositive(0));
}
}
```

Writing a JUnit test class

- This page is really only for expensive setup, such as when you need to connect to a database to do your testing
  - If you wish, you can declare one method to be executed just once, when the class is first loaded

```
@BeforeClass
public static void setUpClass() throws Exception {
      // one-time initialization code
}
```

  - If you wish, you can declare one method to be executed just once, to do cleanup after all the tests have been completed

```
@AfterClass
public static void tearDownClass() throws Exception {
      // one-time cleanup code
}
```

Special features of @Test

- You can limit how long a method is allowed to take
- This is good protection against infinite loops
- The time limit is specified in milliseconds
- The test fails if the method takes too long

- ```
  @Test (timeout=10)
   public void greatBig() {
      assertTrue(program.ackerman(5, 5) > 10e12);
  }
  ```

  - Some method calls should throw an exception
  - You can specify that a particular exception is expected
  - The test will pass if the expected exception is thrown, and fail otherwise

- ```
  @Test (expected=IllegalArgumentException.class)
  public void factorial() {
     program.factorial(-5);
  }
  ```

Ignoring a test

- The @Ignore annotation says to not run a test

```
@Ignore("I don't want Dave to know this doesn't work")
@Test
public void add() {
    assertEquals(4, program.sum(2, 2));
}
```

- You shouldn't use @Ignore unless you have a very good reason!

Homework

1. Demonstrate Proxy pattern in C++!
   - Create a smart, reference counting pointer to any object!
   - Use templates
2. Demonstrate adapter pattern in Java
   - Based on a GUI example:
     - Two different methods of specifying size and location of UI elements
3. Create a Java application to demonstrate JUnit capabilities
   - Basic calculations
   - Populating an array

Objective-C

Next week