Programming Methods Summary

Notation: /forall, /exists, /min, /max, /sum, /product, /num_of(cardinal)
@pre @post @modifies @return @param @throws
In post pui /result == si poate folosesti /old care e valoarea la intrare in functie

Slides1: not monolithic but procedural abstraction(methods)
Divide and Conquer:

1. split problem into subproblems(decomposition)
2. solve subproblems independently(recursion possible)
3. combine subproblem solutions into total solution

Solutions can be methods classes packages etc.
User takes care that pre holds, provider takes care of post and user uses the post.
Contract = interface between user and provider
Procedure = void, Function = non-void
TDD:

1. Analyze requirements
2. Select requirement to develop
3. Specify class/methods informally: javadoc summary
4. Specify formally: model with invariants, headers and contracts
5. Implement rigorous tests, in unit test class
6. Choose data rep and implement class methods
7. Test the implementation and fix defects
8. Repeat

Slides2: Design Principle:
DRY = Don’t Repeat Yourself - avoid code duplication
Abstraction:

- Separate input, calculations and output
- Parameterize methods

Advantages Divide and Conquer:

- enables solution of more complex problems
- organizaes communication about solution domain
- facilitates parallel construction by a team
- plan work and track progress
- verifiability(review, unit testing, stepwise integration)
- maintainability
- reuse

Disadvantages of Divide and Conquer:

- Overhead
- Result looks bigger
- Functions need to be implemented in groups: classes

Functional Decomposition:

- Maximize cohesion: keep related things together, separate unrelated things
- Minimize coupling = minimize dependency
- SRP: Single Responsability Principle = each function should solve a singel well-defined problem
- DRY
- each function should have a small simple and clear implementation: a few dozen lines

Generalization: solving a more general problem than initially required
Generalize by means of paramenters not global variables(GenericIntRelation)
IEEE Classification:

- failure: product deviates from requirements during use
- defect, fault: anomaly in a product that can somehow lead to a failure
- mistake: human action causing a fault
- error: difference between actual and specified result

Detecting defects: reviewing and testing

TESTS:

```
1 @Test(expected = NullPointerException.class)
2 @Test(timeout = 1000)
3 @Before // initialize fixture
4 @After // finalize fixture
5 @Test
6 public void testCase1()
7 @Test
8 public void testCase2()
```

Testing Techniques

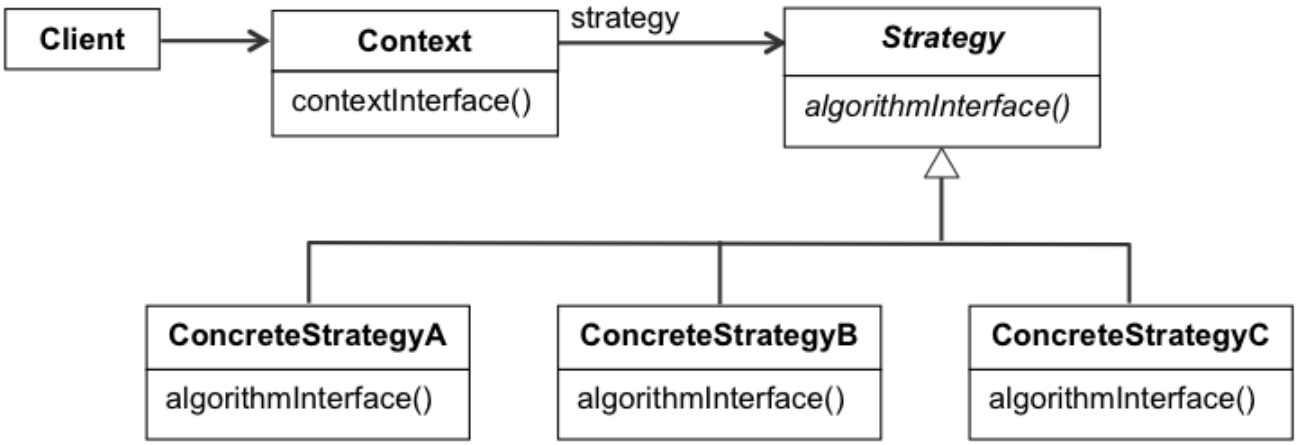
- Boundary analysis
- Equivalence classes
- Statement, branch and path coverage
- Rnadom input

Develop test cases before coding TDD

Design Pattern = outline of a general solution to a design problem, reusable in multiple, diverse, specific contexts

Strategy Design Pattern: accomodate multiple implementations of same method and allow selection of implementation at runtine

Put specification of methods in abstract class or interface
Put implementations in subclasses of spec
Declare variable with specification type
Assign to variable a class of an implementation type




```
1 public abstract class AbstractPowerClass {
2   public abstract long power(int a, int b);
3 }
4 public class PowerA extends AbstractPowerClass{}
5 public class PowerB extends AbstractPowerClass{}
6 public class Context {
7   private AbstractPowerClass myPower;
8   public Context(boolean slow) {
9     if(slow)
10      myPower = new PowerA();
11    else
12      myPower = new PowerB();
13
14    myPower.power(....);
15  }
16 }
```

Slides3

Robustness = Check if pre true. If not throw exception

An Exception signals that something unusual occurred. Exceptions cannot be ignored; the program terminated if an exception is not caught anywhere

```
1 /*
2  * @throws Exception if precondition violated
3  */
4 public void method(int a) throws Exception {
5   if...
6       throw new Exception("Something bad happened");
7 }
```

Testing: When the appropriate exception is thrown the test passes.

When no exception is thrown, the test fails

When the wrong exception is thrown, the test fails.

Also test that there exists an exception message.

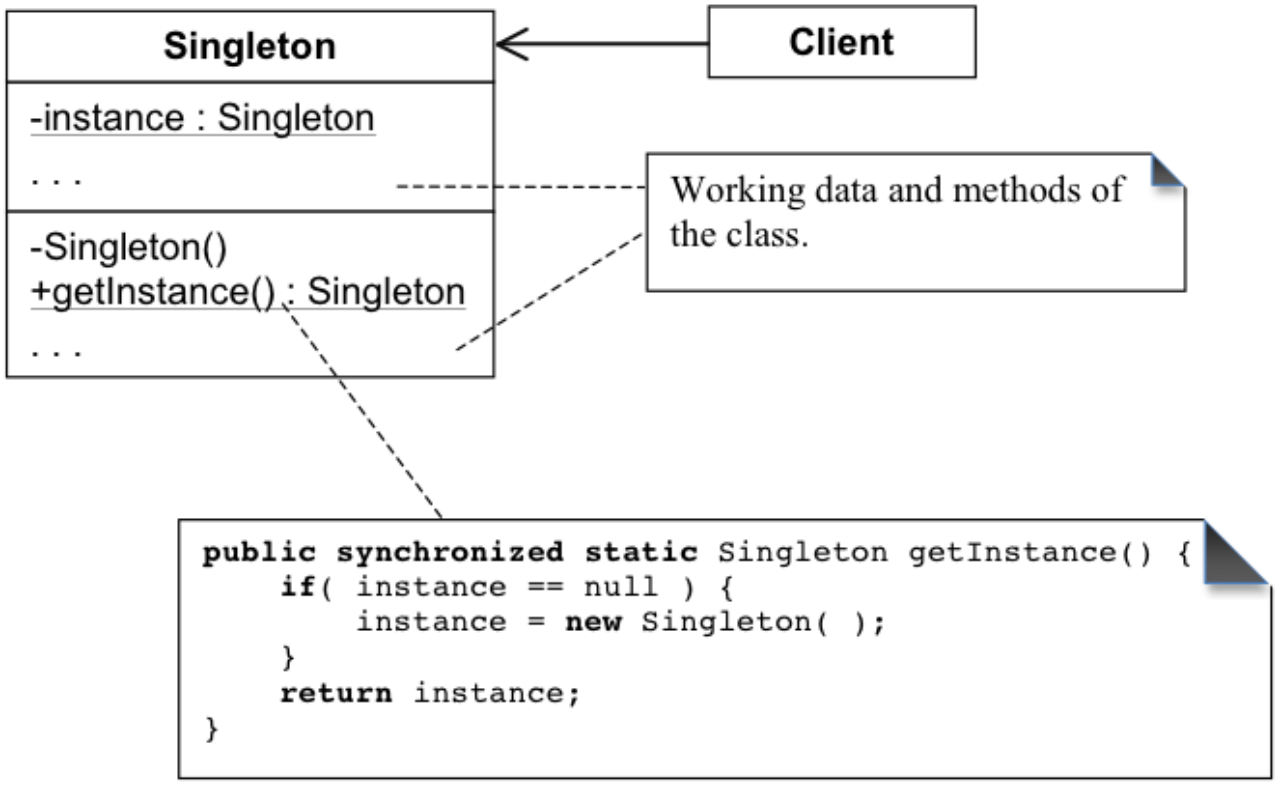
```
1 @Test
2 public void testMethodException() {
3   Class expected = Exception.class;
4   try {
5     method(a);
6     fail("should have thrown " + expected);
```

```
  }
  catch (Exception e) {
    assertTrue("Type; " + e.getClasss().getName()
      + "should be instance of " + expected,
        expected.isInstance(e));
    assertNotNull("message should not be empty", e.getMessage());
  }
}
```

Slides4
Checked Exceptions: Exceptions: used to signal recoverable special situations
ex: FileNotFoundException
Unchecked Exceptions: Runtime Exceptions: used to signal non-recoverable failures
ex: NullPointerException, IndexOutOfBoundsException

Singleton Design Pattern: one Controller, one Undo-Redo facility, one Logger
How to prevent creation of multiple instances?
Singleton ensures that not more than one instance of a class is created and provides a global point of access to this instance

- Make the constructor of the class private
- Add a public static method getInstance()
- The first time getInstance() is called an instance of the class is created, cached and returned
- On subsequent calls, the cached instance is returned




```
public class Logger {
  private static Logger theLogger;
  private Logger() {}
  public static Logger getInstance() {
    if(theLogger == null) {
      theLogger = new Logger();
    }
```

```
    return theLogger;
  }
}
```

Singleton Disadvantages:

- Weakened modularity: other modules will depend on the global object
- Testing is more difficult: tests must also work with that one global object; cannot easily vary it
- Code reuse is hindered: to reuse code depending on the global, you must also reuse that global/object

To avoid dependence on global objects, apply Strategy and invert dependencies.

Slides5: Enumaration Type, Data Type and ADT
Data Abstraction: Combine variables of primitive types that frequently occur together into a single abstraction, instead of handling them separately
ex: numerator and denominator of fraction, coeff of polynomials, a pair of coordinates of a point
Data Type: primitive: int, double, char, boolean
Enumerations: public enum T {NAME1, NAME2, ... , NAMEn}
ex: for(T v: T.values())

ADT: An Abstract Data Type is a type whose specification and usage abstracts from implementation details
The implementation deals with the choice of data representation and method implementations.
The implementation of an ADT can be changed without affecting the usage occurrences, provided it adheres to the ADT's specification
This is called implementation hiding also known as **encapsulation**.
Relate usage to contract and implementation to contract, never together.
ex: List, Set, Map with implementations ArrayList, HashSet and HashMap
Specification of Syntax:
Type name
Operations with names and typed parameters:

- Constructor
- Queries(state inspection) with return type
- Commands(state change), void

```
/**
 * An {@code IntRelation} object maintains a relation on samll integers.
 * The relation is a subset of {@code [0..n) x [0..n)}
 * ....
 *
 * Model: subset of {@code [0..extent()) x [0..extend())}
 *
 * @inv {@code NonNegativeExtend: 0 <= extend()}
 */
public abstract class IntRelation {
  public IntRelation(final int n) {
    //constructor
  }
  public abstract int extent();//query
  public abstract boolean areRelated(int a, int b);//query
  public abstract void add(int a, int b); // command
  public abstract void remove(int a, int b); // command
}
```

Contract ADT:

- Set of (abstract) values
- Contract for individual operations: pre and post in terms of abstract values
- Public invariants: guaranteed relationships between model variables and basic queries

Interface: an interface defines a collection of method headers, with contracts.
Does not have instance variables, nor method implementations
A class can implement one or more interfaces via **implements**.
ADT implementation:

- provide a data representation, a rep invariant and an abstraction function
- provide method implementations adhering to contract

Data rep: instance variables that represent intended abstract values of the ADT
Rep inv: condition to be satisfied by the instance vars: isRepOk()
ex: NotNull: relation != null, Extent: , ElementsNotNull, ElementsSameSize
Abstraction function: maps each representation that satifies the rep invar5iant to the represented abstract value
//AF(this) =. set of (a,b) such that relation[a][b] holds

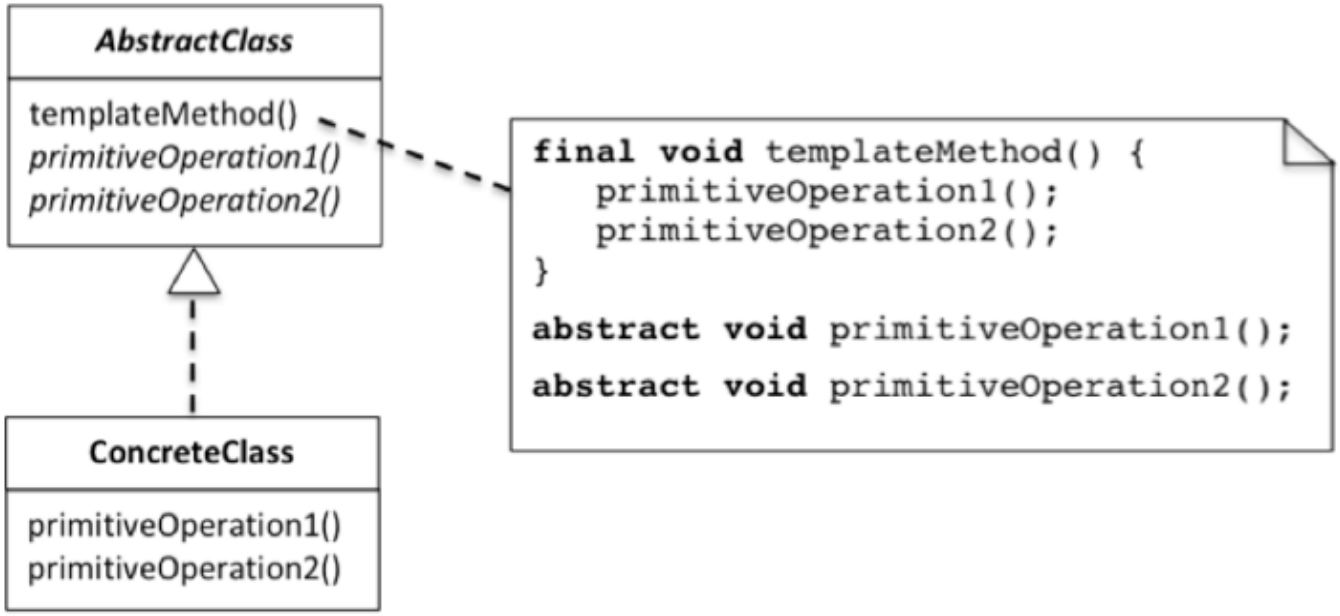
Slides6:
Mutability
Data Type T implemented as a class is said to be immutable when non of its methods modifies this or a parameter of type T = the state of a T obkect cannot change after construction
It is mutable otherwise
Arrays are partly immutable(length), partyly mutable(items)
Two objects are:

- equal if every sequence of operations(queries and commands) applied to both objects yields the same result
- similar if every sequence of queries only applied to both objects yields the same result

Mutable objects are equal when they are the same object
Immutable objects are equal when they are similar

Template Method Design Pattern
Some code fragments may resemble each other, without being duplicates
Resemblence is in overall structure, difference is in some steps
Templated:

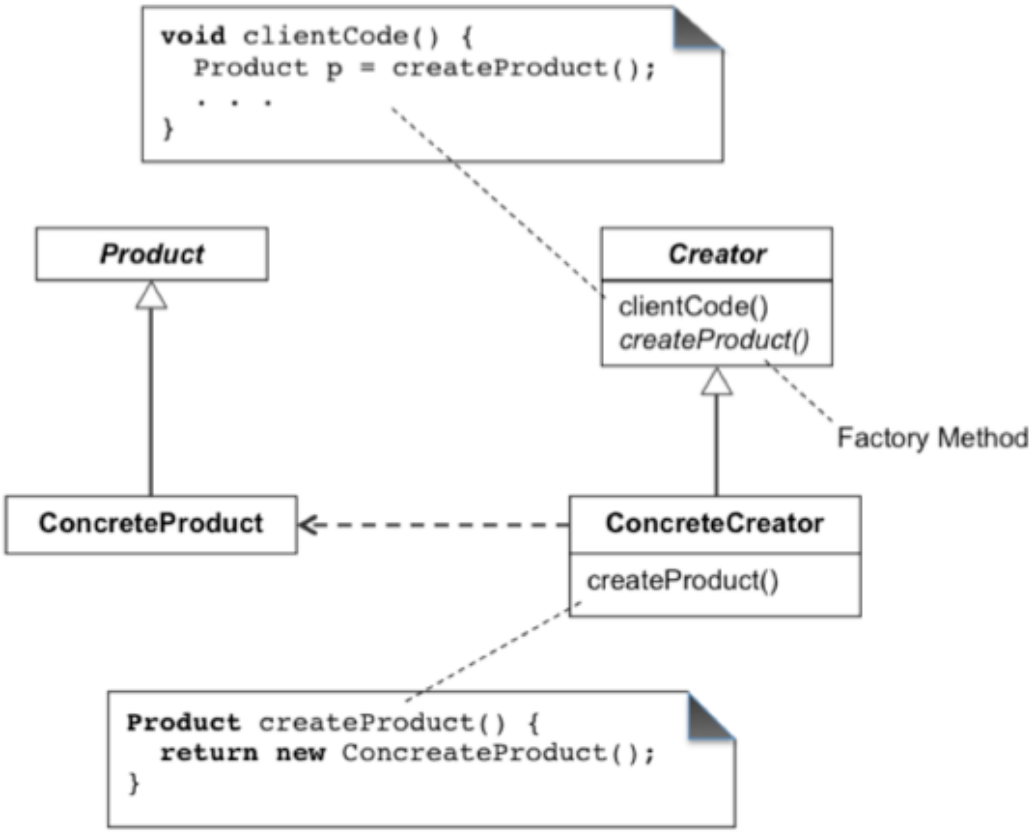
- the structure of an algorithm is represented once
- with variations on the algorithm implemented by subclasses
- The skeleton of the aglorithm is declared in a templated method in terms of overridable operations(hook methods)



Factory Method Design Pattern

- ***new*** statement requires constructor of a concrete class: Set<Node> visited = new HashSet<>();
- This makes client code depend on the concrete class
- Harder to test and reuse
- Violated Dependency Inversion Principle(DIP)
- Program to an interface not to an implementation

Factory defines an interface for creating an object but lets subclasses decide which class to instantiate



Slides7
Operation on Iterable<T> E
Iterator<T> iter = E.iterator();

iter.hasNext();
T v = iter.next();

```
1 public interface Iterable<T> {
2   Iterator<T> iterator();
3 }
4 public interface Iterator<T> {
5   boolean hasNext();
6   T next();
7   void remove() // optional
8 }
```

Iterator Design Pattern

- given a class that manages a collection, provide a way to traverse the elements of the collection such that code for iterating is separate from and looselt coupled both to client code and to the collection
- Iterator<E>: holds state of one specific iteration over E, cannot be reused after the iteration terminates
- Iterable<E> represents a collection over E that can only be iterated over with for-each statement
- can be reused for multiple iterations
- works via its iterator() that returns an Iterator<E>

Collection class must implement interface Iterable<T> and define method iterator() returning an Iterator<T>
Multiple iterator over the same collection can be active at the same time
Testing:

- Check that every element is returned at least once
- Check that every element is returned at most once
- Check that it works without calls to hasNext()
- Check that it works with multiple calls to hasNext()

Slides8
Each compilation unit defines one public class and, next to it possibly other non-public classes, called top-level classes.
A class can also be defined inside another class = Nested classes

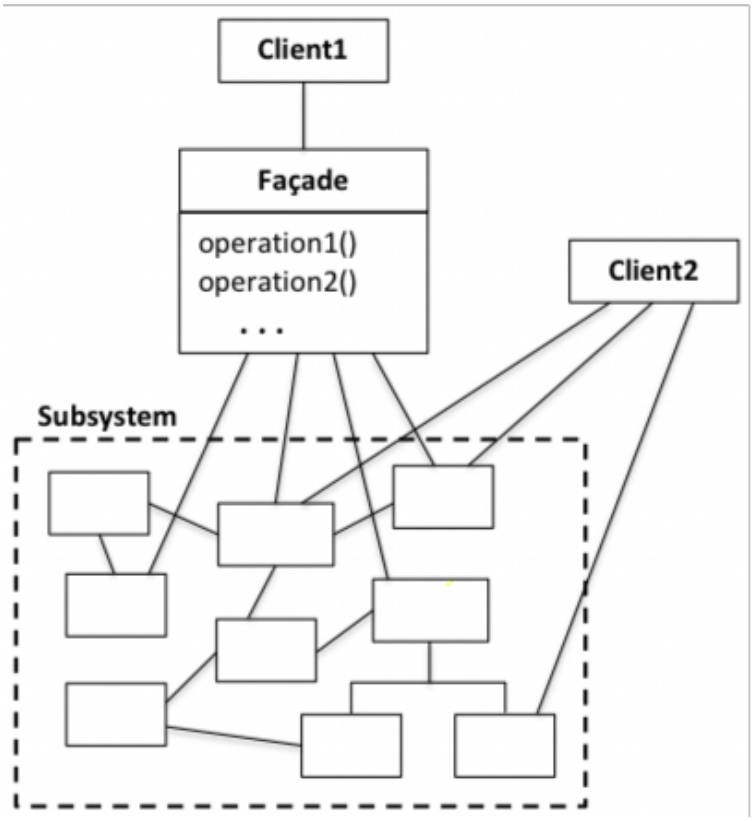
- static member class: almost equivalent to top-level class
- non-static member class
- named local class: defined inside method
- anonymous class: defined in new expression without name

The latter three are called inner classes that have access to the members of its outer classes.
Advantages of nested class: keeps things close together, encapsulation(decreased coupling), improved readability and maintainability of source code, simpler code
Disadvantage: outer class becomes less readable

Generic Type: List<E>
Substitute concrete class type for formal type parameter: List<String> = new ArrayList<>();

Facade Design Pattern
A class or package of classes offers a large complex interface
Many clients of this class do not need all functionality.
Intent:

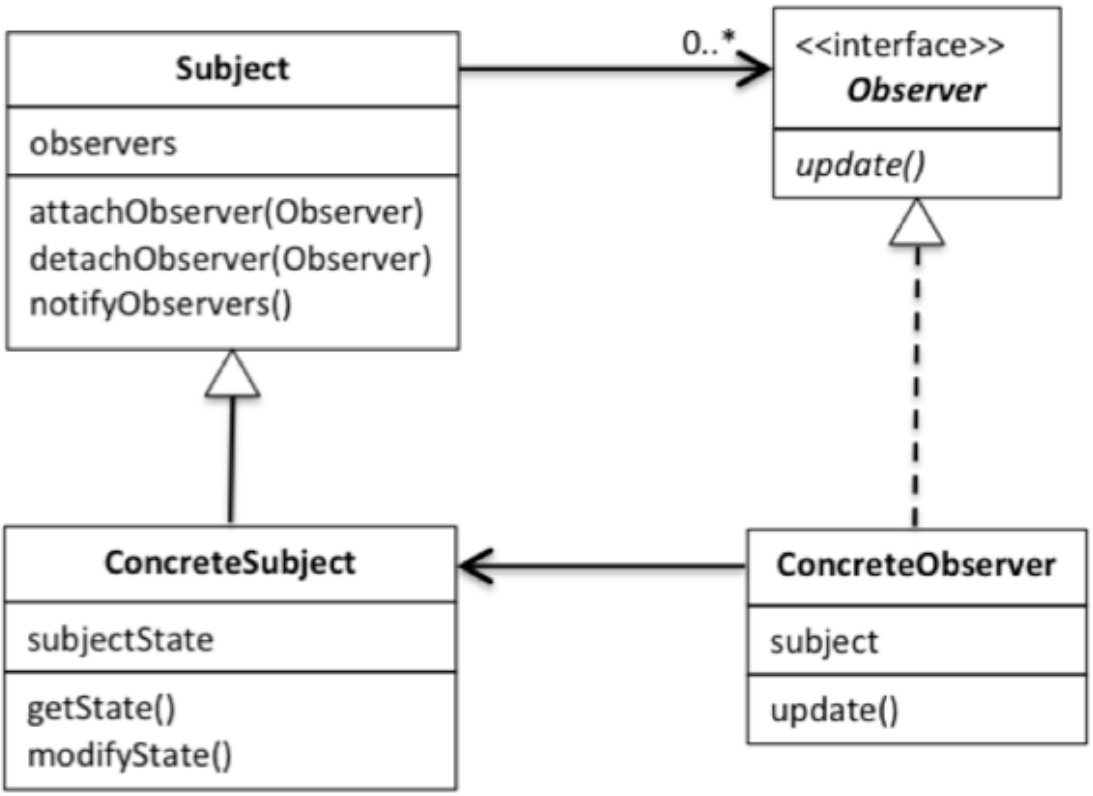
- you offer a single point of access for clients
- you offer a simpler, more abstract, interface for clients
- you decouple client code from multiple subsystem classes



Slides9
Callbacks = provide one or more parameter to the computation(not possible in Java pre 8 and in Java 8 only using lambda expressions)
In Java we need to pass an object as carrier for callback method
Generator with listener interface:
GUI provides listener object to Generator. Listener object implements event method objectGenerated. Generator calls event method of listener object. => no cyclic dependency
So, in generator call when we have a solution listener.objectGenerated(solution)
We can also give the listener in the constructor of the generator.
We can also have a method setListener(GeneratorListener listener).
We can also have multiple listeners in a List with functions addListener and notifyListeners which is the function we call in generate.

Observer Design Pattern:

- Subject: defines interface for observables
- ConcreteSubject: implements Subject interface
- Observer: defines interface for events that can be observed
- ConcreteObserver: implements Observer interface to handle the observable events
- Client: configures concrete subject with concrete observers



Push versus Pull: Decision on what data to transfer
Push to Observers: Subject Decides
Notify Observer => Pull from Subject: Observer Decides
Can also use a combination of both: Push a part of data that you think all observers need and let extra data be pulled by the observers when needed

Slides10
Dependency Inversion Principle:
Module A depends on module B: A cannot be compiled used without having B, hindering testing and reuse of A
DIP = high-level modules should not depend upon low-level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions
Using the callback interface via a Parameter

```
1 public interface FunctionalityB{
2   void doB(String data);
3 }
4 public class FunctionalityA {
5   public void doA(int n, FunctionalityB b){
6     // do something
7     b.doB(data);
8   }
9 }
10 public class DataPrinter implements FunctionalityB {
11   @Override
12   public void doB(String data){
13     //do something with data
14   }
15 }
16 FunctionalityB printer = new DataPrinter();
17 FunctionalityA.doA(n,printer); // Dependency Infection on THIS LINE
```

If we have a class with another interface:

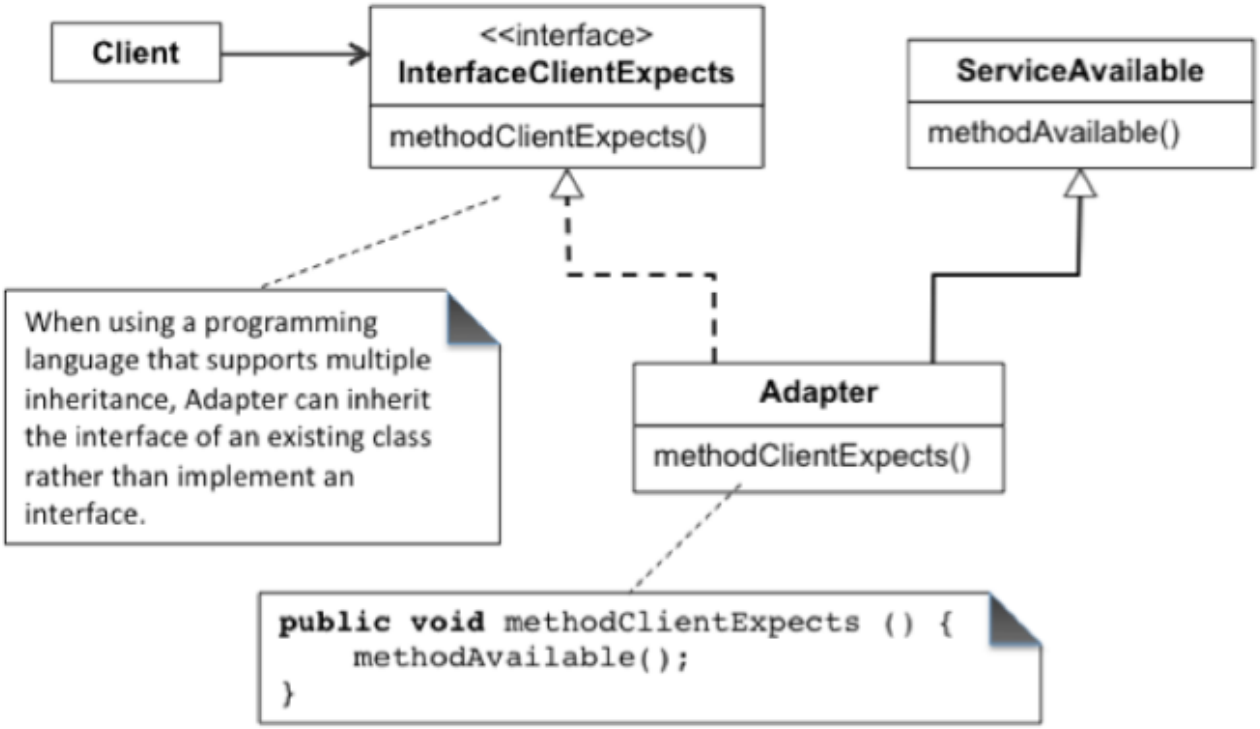
class DataCounter extends Counter implements FunctionalityB
Here implement doB using functions from Counter
Disadvantage of this: Adapted class DataCounter has hard coupling to implementation from Counter, thus it cannot be used to adapt SpecialCounter which extends Counter. You would need to define SpecialDataCounter for that thus leading to duplicated code.

Do this via composition:

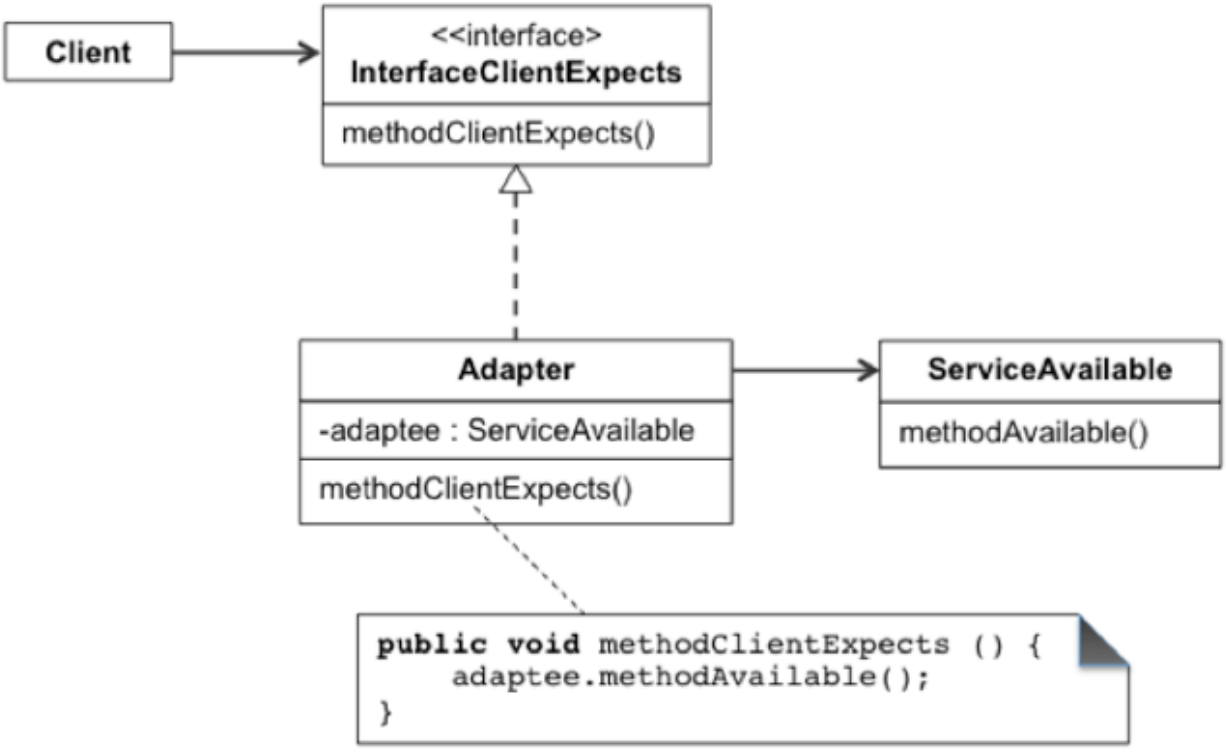
```
class CounterAdapter implements FunctionalityB {
  private final Counter counter;
  public CounterAdapter(Counter counter) {
    this.counter = counter;
  }
  @Override
  public void doB(String data) {
    counter.count();
  }
}
//Use it like this
final Counter counter = new Counter();
final CounterAdapter = new CounterAdapter(counter);
Functionalitya.doA(n,adapter);
```

Composition requires extra overhead from using inheritence but comes with the benefit of extra indirection doB to count via counter object.
This buys you more independence and flexibility
Can also have have a composite functionalityB implemented using a list
of FunctionalityB that calls doB for every element of the list.

Adapter Design Pattern
The Adapter DP is useful in situations where an existing class provides a needed service but there is a mismatch between the interface offered and interface that clients expect
The Adapter shows how to to convert the interface of the existing class into the interface that clients expect
Using Inheritence: creates tigether coupling, to a specific implementation
Gives compile-time dependency

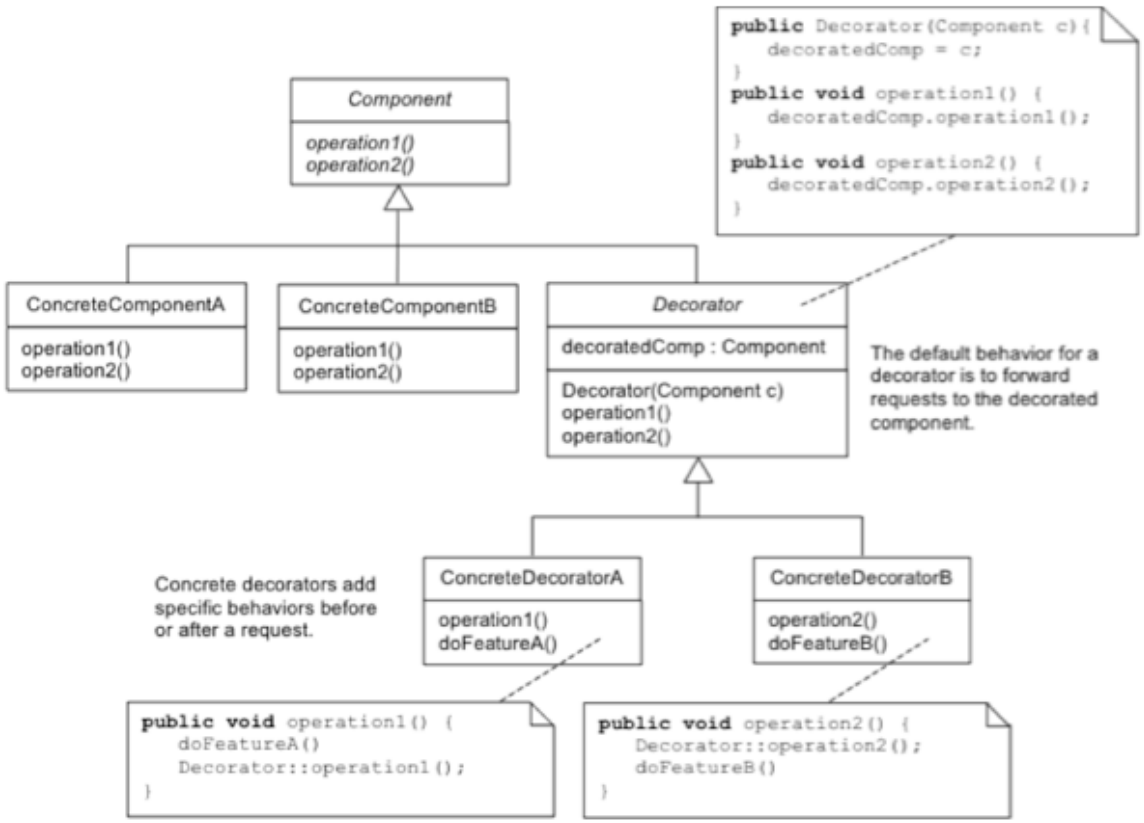


Using Composition: offers looser coupling: couples to an object



Decorator Design Pattern

The decorator design pattern provides a way of attaching additional responsabilities to an object dynamically. It uses object composition rather than class inheritance for a lightweight flexible approach



Adapter and Decorator differences:

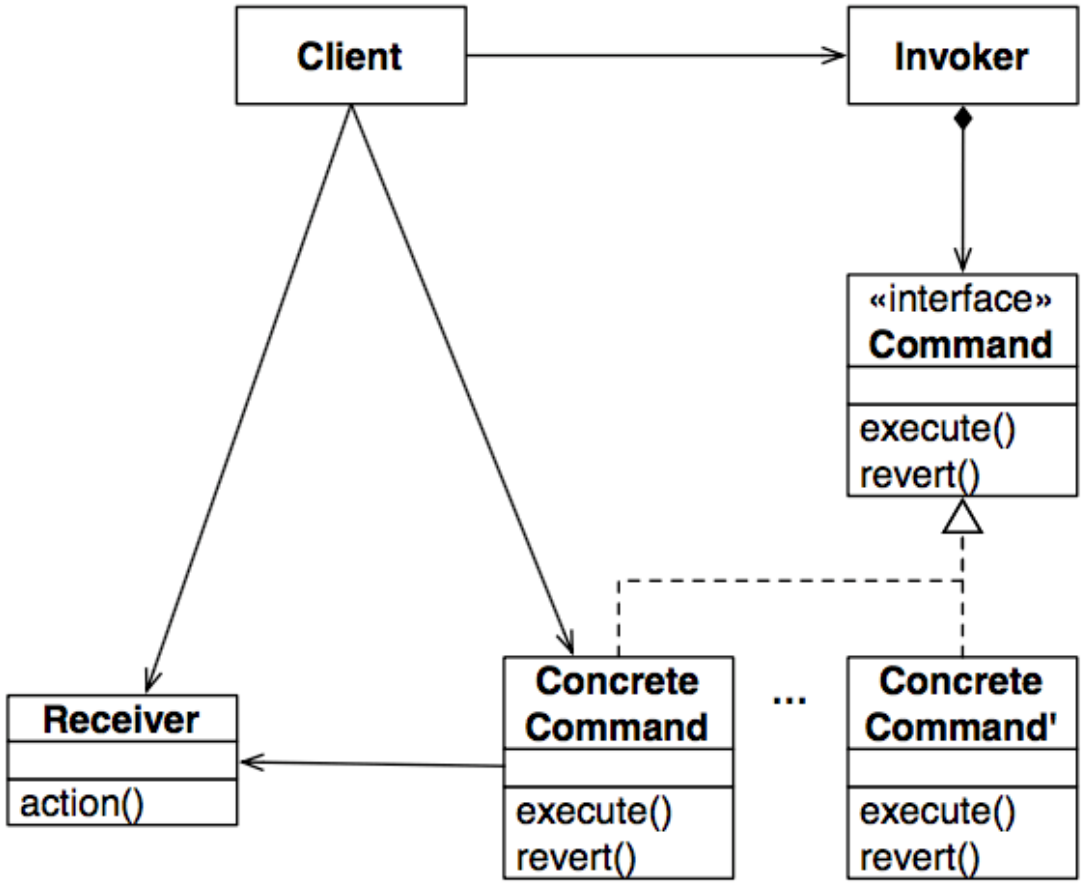
- Interface: Adapter - differs from adapted; Decorator - same as decorated
- Functionality: Adapter - same as adapted; Decorator - differs from decorated

Slides11

Command Design Pattern

Command class encapsulates all data to call a method elsewhere:

- which method to call, including in which object: Receiver
- which actual parameters to supply as arguments
- what to do with the return value



Compound Commands consist of a sequence of commmands(stored in a list maybe). It provides a method to add commands.
The execute method executes the commands in order
The revert method reverts the commands in reverse order

Slides12
Major concerns for software developer:

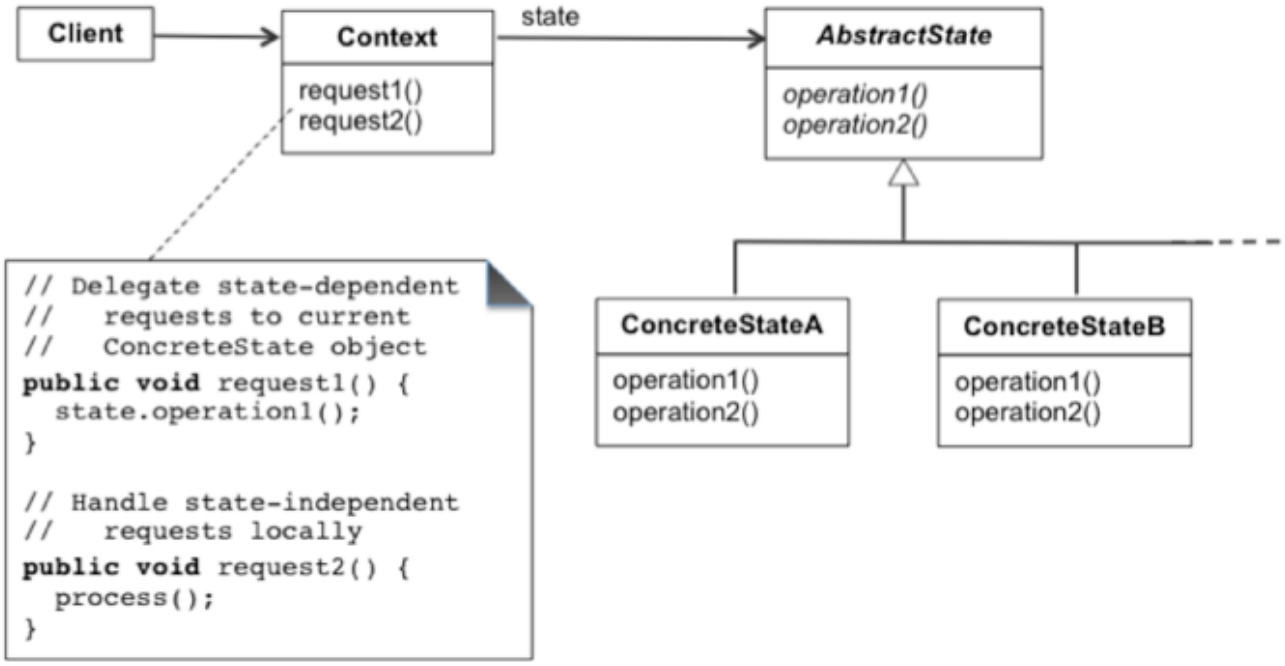
- functional correctness
- performance: speed, memory
- verifiability
- maintainability
- reuseability

Design Patters help with the last three points

State Design Pattern
Multiple states/modes: with same events, different behaviors
How to prevent clumsy conditional selection of behavior
Intent: If an object goes through clearly identifiable states/modes, and the object's behavior is especially dependent of its state, then it is a good candidate for the State Design Pattern
Solution:

- Superclass of interface specifies all events handled by a state
- The concrete event handlers are implemented in concrete states. Different states can handle the same event in different ways
- Context code: holds a current state object, delegates event handling to the current state object, decides on

change of state



Open Closed Principle(OCP)
Two ways of reusing a class:

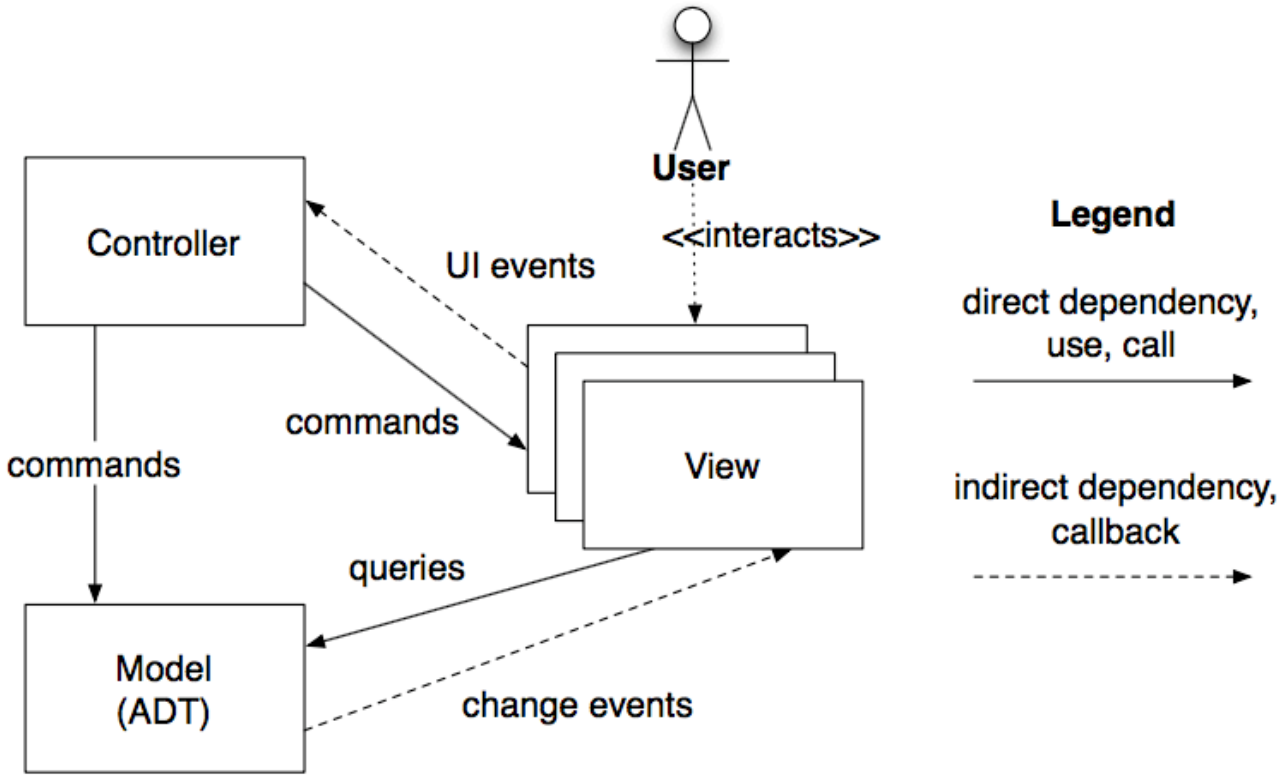
1. By regular client code: Instantiate new objects, and call methods on them
2. By code in (possibly anonymous) subclass that extends the class: access super functionality

Open close principle:

- Modules should be open for extension via inheritance
- Modules should be closed for modification by clients: private or at least protected instance variables

Use Adapter or Decorator Design Patterns to preserve this principle
Also the State DP is a way of adhering to OCP: add subclass for new state with new event behaviors

Model-View-Controller Architecture:



Slides13
Concurrency: to decouple GUI from long running computations
Using SwingWorker
SwingWorker acts as a Facade for the Java thread facilities

- Which computation to do in background: doInBackground()
- How to handle intermediate results: process()
- What to do with the final result: done()

Interface Segregation Principle(ISP):
When designing a class for several clients with different needs, rather than loading the class with all methods that clients need and making each client depend on the complete interface, create specific interfaces for each king of client.
Make each client depend only on its interface
Implement all interfaces in the class

Composite Design Pattern:
It organizes objects into a tree data structure where lead nodes represent individual objects and interior nodes present compositions of individual objects and/or other compositions