Programming Language Design

12/16 Aspect-Oriented Programming, Event-Driven Programming, and Context-Oriented Programming

Suppose we have such a system

- Shopping platform
  - Customers can find and buy the items they want
  - When a customer clicked
    - "Add to Cart"
    - "Proceed to Checkout"
- Backend
  - Handle payment and update its database

To implement its backend

- Main logic
  - In stock?
    - Charge a credit card
    - Put it on layaway
  - Is it enough? No!
    - Check if this operation is authorized
    - Ensure the transaction is completed
    - Lock for data access
    - Log for debugging

With OOP, you can…

- Implement a class for main logic
  - Manage the data and provide operations such as “checkout”

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {


        /* in stock? */


        /* charge the credit card */
        /* put the item on layaway */



    }
}
```

With OOP, you can… (cont.)

- Implement a class for authorization
  - Authorize this operation at the beginning of “checkout”

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {
        /* authorization */

        /* in stock? */



        /* charge the credit card */
        /* put the item on layaway */



    }
}
```

With OOP, you can… (cont.)

- Implement a class for transaction
  - Ensure "charge" and "put on layaway" be atomic

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {
        /* authorization */

        /* in stock? */
        /* beginning of the transaction */

        /* charge the credit card */
        /* put the item on layaway */

        /* end of the transaction */

    }
}
```

With OOP, you can… (cont.)

- Use a lock class provided by the library
  - Lock for concurrency control

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {
        /* authorization */
        /* lock for thread-safety */
        /* in stock? */
        /* beginning of the transaction */

        /* charge the credit card */
        /* put the item on layaway */

        /* end of the transaction */
        /* unlock for thread-safety */
    }
}
```

With OOP, you can… (cont.)

- Use a logger class provided by the library
  - For tracing and debugging

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {
        /* authorization */
        /* lock for thread-safety */
        /* in stock? */
        /* beginning of the transaction */
        /* log the start of this operation */
        /* charge the credit card */
        /* put the item on layaway */
        /* log the completion of this operation */
        /* end of the transaction */
        /* unlock for thread-safety */
    }
}
```

Usually it looks like…

- Method calls to well-defined classes
  - So far so good

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {
        AccountMgr.authorize(a, i);
        objLock.lock();
        /* in stock? */
        transMgr.begin(a, i, c, …);
        Logger.info("start:", this, a, I, c, …);
        /* charge the credit card */
        /* put the item on layaway */
        Logger.info("completion:");
        transMgr.commit();
        objLock.unlock();
    }
}
```

Is it modularized enough?

- Not all statements belong to this concern…
  - DataMgr::checkout() should only contain the code for “checkout”!

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {
        AccountMgr.authorize(a, i);
        objLock.lock();
        /* in stock? */
        transMgr.begin(a, i, c, …);
        Logger.info("start:", this, a, I, c, …);
        /* charge the credit card */
        /* put the item on layaway */
        Logger.info("completion:");
        transMgr.commit();
        objLock.unlock();
    }
}
```

Suppose we want to change the log

– We have to change the code that are scattered in many places

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {
        AccountMgr.authorize(a, i);
        objLock.lock();
        /* in stock? */
        transMgr.begin(a, i, c, …);
        Logger.fine("start:", this, a, I, c, …);
        /* charge the credit card */
        /* put the item on layaway */
        Logger.fine("completion:");
        transMgr.commit();
        objLock.unlock();
    }
}
```

Wait a minute!

- Can't we use logging levels to resolve this problem?

- Assign every log message a level
  - at compile-time
- Set a system-wide level

→ Print a message if its level >= the system-wide level

  - at runtime

For example, java.util.logging

- java.util.logging.Level
  - OFF (Integer.MAX_VALUE) > SEVERE (1000) > WARNING (900) > INFO (800) > CONFIG (700) > FINE (500) > FINER (400) > FINEST (300) > ALL (Integer.MIN_VALUE)

```
import java.util.logging.*

Logger logger = Logger.get("my logger");
logger.setLevel(Level.INFO);
logger.info("this will be printed");
logger.fine("this one should not appear!");
```

Can't we use logging levels to resolve?

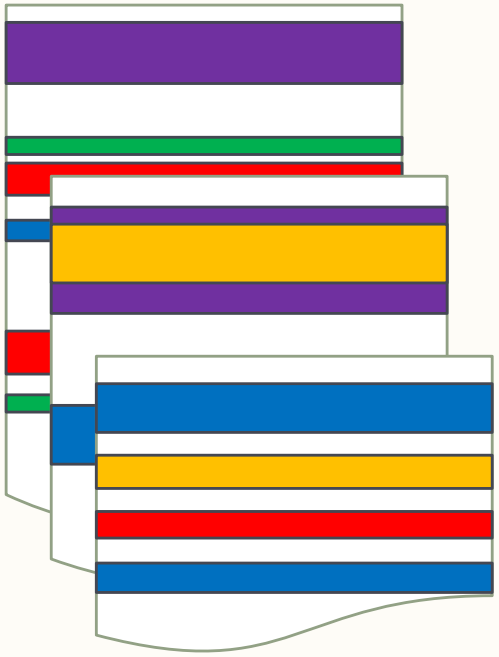
- Obviously NO, since the code for logging is still there
  - Even in C language we can expand macros nothing
- Such code cannot be gathered in a module with only OOP

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {
        AccountMgr.authorize(a, i);
        objLock.lock();
        /* in stock? */
        transMgr.begin(a, i, c, …);
        Logger.fine("start:", this, a, I, c, …);
        /* charge the credit card */
        /* put the item on layaway */
        Logger.fine("completion:");
        transMgr.commit();
        objLock.unlock();
    }
}
```

Crosscutting concern

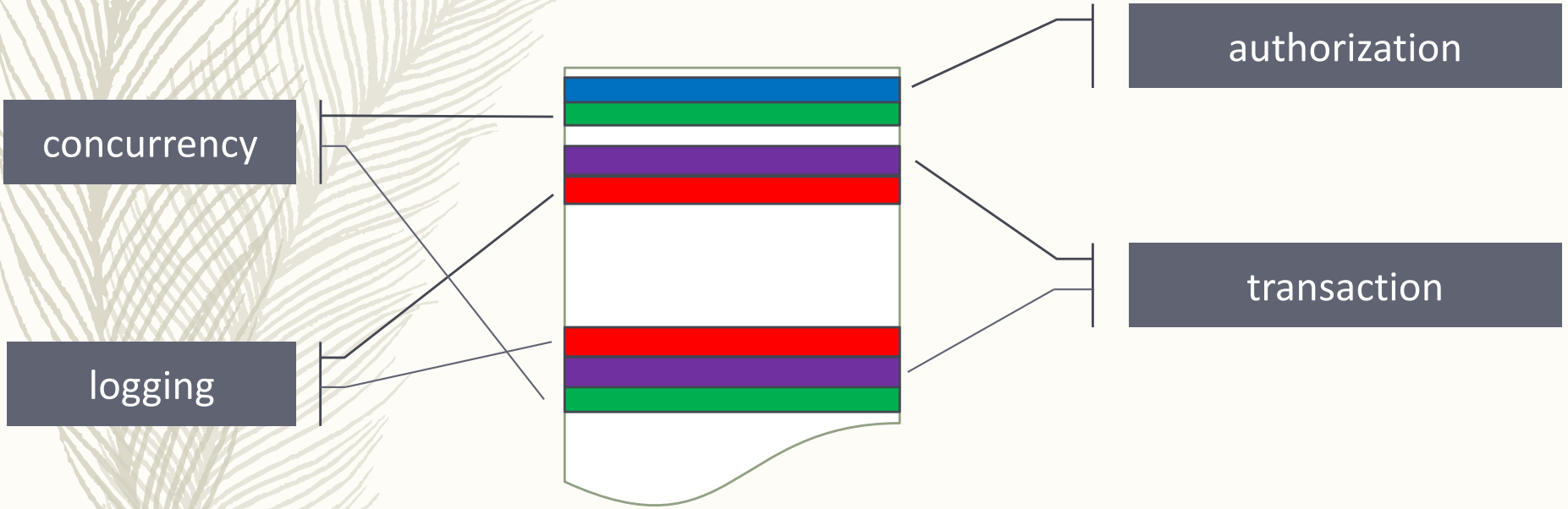
- Such concerns cross-cut the system's basic functionality
  - In this example, logging, transaction, etc.

```
class DataMgr extends Backend {
    boolean checkout(Account a, Item I, Card c, …) {
        AccountMgr.authorize(a, i);
        objLock.lock();
        /* in stock? */
        transMgr.begin(a, i, c, …);
        Logger.fine("start:", this, a, I, c, …);
        /* charge the credit card */
        /* put the item on layaway */
        Logger.fine("completion:");
        transMgr.commit();
        objLock.unlock();
    }
}
```



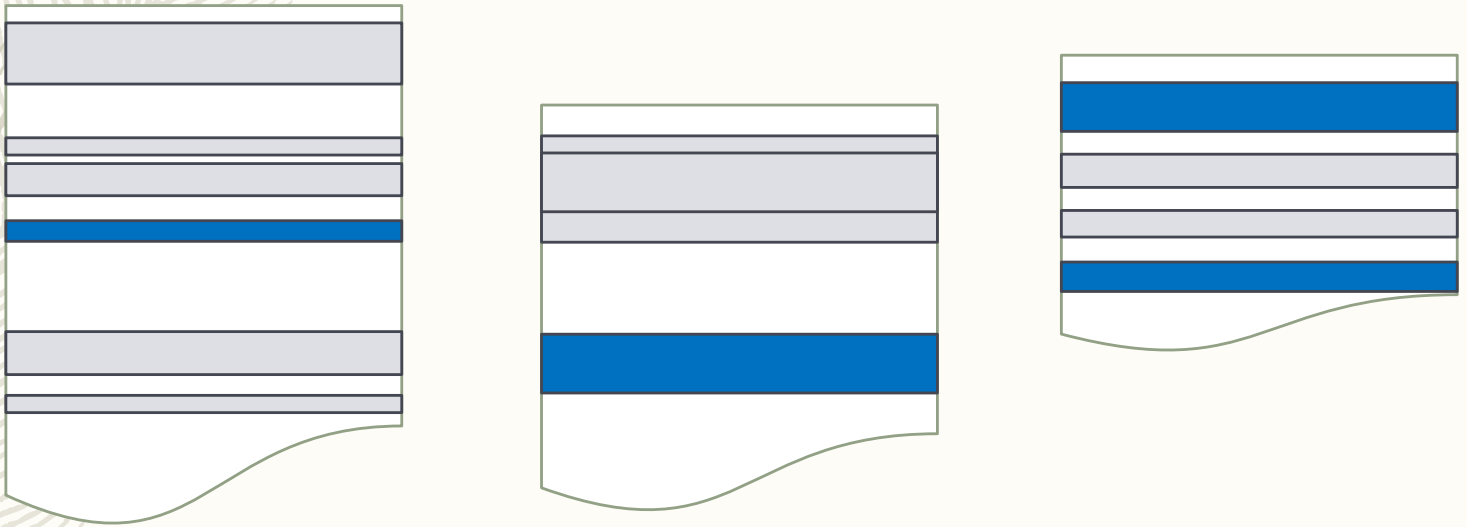
Code tangling

- The codes for different concerns are tangled
  - Not be separated well



Code scattering

- The code for one concern is scattered throughout the program
  - Not be gathered in one module



Design processes and languages

– "Design processes break a system down into smaller and smaller units."

– "Programming languages provide mechanisms that allow the programmer to define abstractions of system sub-units, and then compose those abstractions in different ways to produce the overall system."

Reference

- *G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-m. Loingtier and J. Irwin. Aspect-oriented programming. In the 11st European Conference on Object-Oriented Programming (ECOOP'97), 1997.*

Crosscutting concern (cont.)

– "We have found many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement."

Reference

- *G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-m. Loingtier and J. Irwin. Aspect-oriented programming. In the 11st European Conference on Object-Oriented Programming (ECOOP'97), 1997.*

Recall: Separation of concerns

- "Focus one's attention upon some aspect"
  - "Does not mean ignoring the other aspects"
  - "It is just doing justice to the fact that from this aspect's point of view, the other is irrelevant"

--- Edsger W. Dijkstra

Reference

- *EWD 447: On the role of scientific thought, 1974.*
- *Edsger W. Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982.*

Aspect-oriented programming (AOP)

- An aspect
  - If it can not be cleanly encapsulated in a generalized procedure
  - Tends not to be units of the system's functional decomposition

- "Many existing programming languages, including object-oriented languages, procedural languages and functional languages, can be seen as having a common root in that their key abstraction and composition mechanisms are all rooted in some form of generalized procedure."

Reference

- *G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-m. Loingtier and J. Irwin. Aspect-oriented programming. In the 11st European Conference on Object-Oriented Programming (ECOOP'97), 1997.*

Join point

- Something happens in the execution of a program such as
  - Method invocations
  - Field accesses
  - Object instantiations

    :
- For example,
  when DataMgr::checkout is called
  when a DataMgr object is created

Join point: call

– When a method is called

```
class Test {

    void test() {

        dataMgr.checkout(…);
    }
}
```

Join point: execution

– When a method body executes

```
class DataMgr extends Backend {

    boolean checkout(…) {
        :
    }
}
```

Join point: object creation

– When an object is created

```
class DataMgr extends Backend {
    Logger logger = new Logger();
}
```

Join point: field accss

– When a field is read

```
class DataMgr extends Backend {
    int serial;
    boolean checkout(…) {
        :
        int s = this.serial;
        :
    }
}
```

Join point: field access (cont.)

– When a field is written

```
class DataMgr extends Backend {
    int serial;
    boolean checkout(…) {
        :
        this.serial = s;
        :
    }
}
```

Pointcut and advice

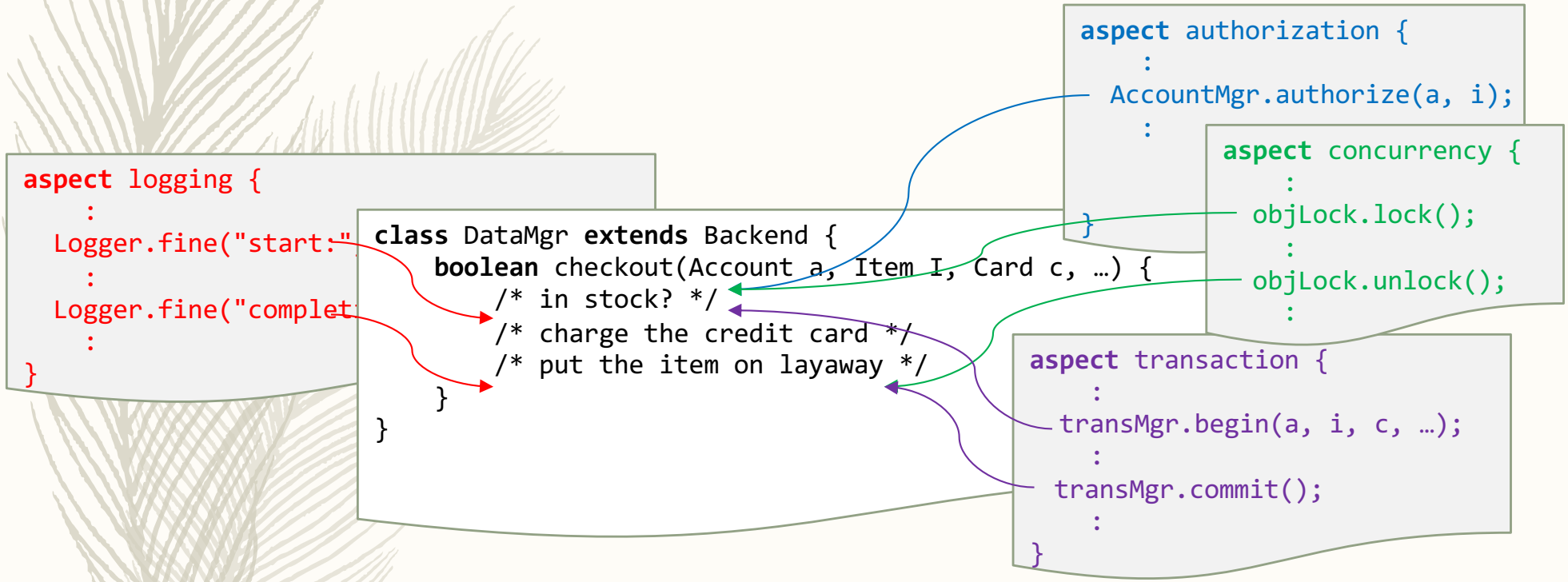
- The ones we use to pick out the join points we care
  - Supported by language constructs
- In order to do some handling at those join points
  - E.g. for the authorization concern we want to pick out when DataMgr.checkout is executed in order to perform AccountMgr.authorize
  - Such a handling is called advice

How AOP works for us?

- For a crosscutting concern, we can
  - Use pointcuts to pick out join points
  - Write the implementations with advice
  - Put pointcuts and advice into an aspect
- Then the aspect weaver will produce a woven system for us
  - Weave the classes and aspects together
    - *At compile-time, load-time, or runtime*

How AOP works for us? (cont.)

– The checkout example looks like this



AspectJ and AJDT

- A general-purpose aspect-oriented extension to Java
- Provide language constructs for aspects, pointcuts, and advice
  - Sophisticated and expressive
- AspectJ Development Tools (AJDT)
  - Provides Eclipse platform based tool support

Reference

- *The AspectJ Programming Guide*
  https://eclipse.org/aspectj/doc/next/progguide/index.html
- *AspectJ Development Tools (AJDT)*
  http://www.eclipse.org/ajdt/

The 101 example: logging

```
// Point.java
public class Point {
  private int x;  private int y;
  public Point(int x, int y) {
    this.x = x;  this.y = y;
  }
  public void move(int dx, int dy) {
    this.x += dx;  this.y += dy;
  }
  public void moveTo(int x, int y) {
    this.x = x;  this.y = y;
  }
  public static void main(String[] args) {
    Point p = new Point(5, 5);
    p.move(5, 5);
    p.moveTo(15, 15);
  }
}
```

- Suppose we have such a Point class
  - Has the fields x and y
  - Has the method move and moveTo

→ if want to log the motion…

The 101 example: logging (cont.)

– And we want to log for the motion

```
// Point.java
public class Point {
  public int x;  public int y;
  public Point(int x, int y) {
    this.x = x;  this.y = y;
  }
  public void move(int dx, int dy) {
    this.x += dx;  this.y += dy;
  }
  public void moveTo(int x, int y) {
    this.x = x;  this.y = y;
  }
  public static void main(String[] args) {
    Point p = new Point(5, 5);
    p.move(5, 5);
    p.moveTo(15, 15);
  }
}
```

aspect

pointcut

```
// Logging.aj
public aspect Logging {
  pointcut motion():
    execution(void Point.move(int, int));

  after(): motion() {
    System.out.println("a point is moved!");
  }
}
```

advice

```
a point is moved!
```

after advice and before advice

- For the join points selected by a pointcut, we can attach advice
  - after the join point

```
after(): yourpointcut() {
        :
}
```

  - before the join point

```
before(): yourpointcut() {
        :
}
```

Binary operators for pointcuts

- && operator (binary)
  - Select join points matching **both** of the pointcuts
  - Intersection
- || operator (binary)
  - Select join points matching **either** of the pointcuts
  - Union

The 101 example: logging (cont.)

– We can select both Point.move and Point.moveTo by using || operator

```
// Point.java
public class Point {
  private int x;  private int y;
  public Point(int x, int y) {
    this.x = x;  this.y = y;
  }
  public void move(int dx, int dy) {
    this.x += dx;  this.y += dy;
  }
  public void moveTo(int x, int y) {
    this.x = x;  this.y = y;
  }
  public static void main(String[] args) {
    Point p = new Point(5, 5);
    p.move(5, 5);
    p.moveTo(15, 15);
  }
}
```

pointcut composition

```
// Logging.aj
public aspect Logging {
  pointcut motion():
    execution(void Point.move(int, int))
    || execution(void Point.moveTo(int, int));

  after(): motion() {
    System.out.println("a point is moved!");
  }
}
```

```
a point is moved!
a point is moved!
```

The 101 example: logging (cont.)

– Selecting field accesses of Point.x and Point.y is also possible

```
// Point.java
public class Point {
  private int x;  private int y;
  public Point(int x, int y) {
    this.x = x;  this.y = y;
  }
  public void move(int dx, int dy) {
    this.x += dx;  this.y += dy;
  }
  public void moveTo(int x, int y) {
    this.x = x;  this.y = y;
  }
  public static void main(String[] args) {
    Point p = new Point(5, 5);
    p.move(5, 5);
    p.moveTo(15, 15);
  }
}
```

```
// Logging.aj
public aspect Logging {
  pointcut motion():
    set(int Point.x) || set(int Point.y);

  after(): motion() {
    System.out.println("a point is moved!");
  }
}
```

```
a point is moved!
a point is moved!
a point is moved!
a point is moved!
a point is moved!
a point is moved!
```

Wildcards and the unary operator

- Wildcards “*” and “..” can be used
  - `pointcut motion(): execution(* Point.move(..));`
  - `pointcut anyexec(): execution(* *(..));`
- ! Operator
  - Select join points except those specified by the pointcut

The 101 example: logging (cont.)

- Exclude the case that setting Point.x and Point.y from the constructor
- Use operator ! and operator &&

```
// Point.java
public class Point {
  private int x;  private int y;
  public Point(int x, int y) {
    this.x = x;  this.y = y;
  }
  public void move(int dx, int dy) {
    this.x += dx;  this.y += dy;
  }
  public void moveTo(int x, int y) {
    this.x = x;  this.y = y;
  }
  public static void main(String[] args) {
    Point p = new Point(5, 5);
    p.move(5, 5);
    p.moveTo(15, 15);
  }
}
```

control flow

```
// Logging.aj
public aspect Logging {
  pointcut motion():
    set(int Point.*)
    && !cflow(call(Point.new(..)));

  after(): motion() {
    System.out.println("a point is moved!");
  }
}
```

constructor

```
a point is moved!
a point is moved!
a point is moved!
a point is moved!
```

this, target, and args

- Used to match the actual type
  - The type selected by other pointcuts are apparent type
- this
  - Match the join points where **this** is an instance of the specified type
- target
  - The same as this but matching target object
- args
  - Select join points based on the argument object's actual type

An example of target pointcut

```
public class Animal {
  public void hello() {
    System.out.println("...");
  }
}
public class Duck extends Animal {
  public void hello() {
    System.out.println("quack!");
}
public class Cat extends Animal {
  public void hello() {
    System.out.println("meow~");
  }
  public static void main(String[] args) {
    Cat c = new Cat();  c.hello();
    Animal a = c;  a.hello();
    a = new Duck();  a.hello();
  }
}
```

```
// Testtarget.aj
public aspect Testtarget {
  pointcut sayHello(): call(void Animal.hello())
                       && target(Cat);
  before(): sayHello() {
    System.out.print("a cat says hello: ");
  }
}
```

Specify a type

```
a cat says hello: meow~
a cat says hello: meow~
quack!
```

The arguments of advice

```
public class Animal {
  public void hello() {
    System.out.println("...");
  }
}
public class Duck extends Animal {
  public void hello() {
    System.out.println("quack!");
  }
}
public class Cat extends Animal {
  public void hello() {
    System.out.println("meow~");
  }
  public static void main(String[] args) {
    Cat c = new Cat();  c.hello();
    Animal a = c;  a.hello();
    a = new Duck();  a.hello();
  }
}
```

```
// Testtarget.aj
public aspect Testtarget {
  pointcut sayHello(Cat c): call(void Animal.hello())
                            && target(c);
  before(Cat c): sayHello(c) {
    System.out.print(c + " says hello: ");
  }
}
```

Specify an object identifier

```
thistarget.Cat@85ede7b says hello: meow~
thistarget.Cat@85ede7b says hello: meow~
quack!
```

around advice

- Surround join points
  - To do something before and after a method
  - To bypass the original implementation
- To execute the original one
  - Call **proceed** inside

Example: Fibonacci

```
// Simple.java
public class Simple {
  public int fib(int n) {
    if (n < 2)
      return n;
    else
      return fib(n - 1) + fib(n - 2);
  }
  public static void main(String[] args) {
    Simple s = new Simple();
    System.out.println(s.fib(40));
  }
}
```

- Fibonacci number
  - 0, 1, 1, 2, 3, 5, 8, 13, …
  - i.e. fib(n) = fib(n - 1) + fib(n - 2)
- So, to calculate fib(40)
  - Calculate fib(39)
    - *Calculate fib(38)*
    - *Calculate fib(37)*
  - Calculate fib(38)

→ Apply cache to make it faster!

Example: Fibonacci with cache

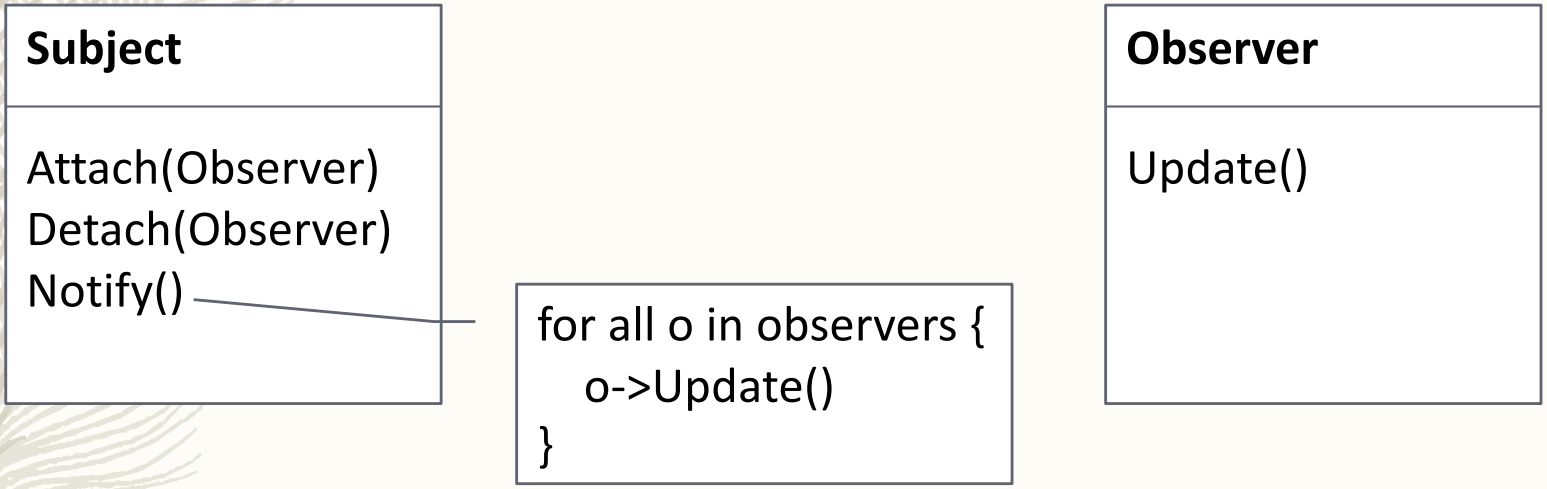
```
// Simple.java
public class Simple {
  public int fib(int n) {
    if (n < 2)
      return n;
    else
      return fib(n - 1) + fib(n - 2);
  }
  public static void main(String[] args) {
    Simple s = new Simple();
    System.out.println(s.fib(40));
  }
}
```

– Use around advice to override the behavior

```
// Cache.aj
public aspect Cache {
  static private int CACHE_SIZE = 50;
  private int[] cache = new int[CACHE_SIZE];
  pointcut fibCall(int n):
    call(int Simple.fib(int)) && args(n);
  int around(int n): fibCall(n) {
    int value = 0;
    if (n < CACHE_SIZE) {
      value = cache[n];
    }
    if (value > 0)
      return value;
    value = proceed(n);
    if (n < CACHE_SIZE)
      cache[n] = value;
    return value;
  }
}
```

Recall: the Observer pattern

- Subject maintains an observer list
  - To remember which observer should be notified
- Observers can register themselves
  - To get notifications from subject



- One of the most widely-used pattern in GoF design patterns

The Observer pattern in Java 8

```
// Subject.java
interface Subject {
  ArrayList<Observer> observers
    = new ArrayList<Observer>();
  default void attach(Observer o) {
    observers.add(o);
  }
  default void detach(Observer o) {
    observers.remove(o);
  }
  default void doNotify(String s) {
    observers.forEach(o -> o.update(s));
  }
}

// Observer.java
interface Observer {
  default void update(String s) {
    System.out.println(this
      + " is updated for " + s);
  }
}
```

- Define interfaces for Subject and Observer
  - Use interfaces as “mixins” in Java 8
- Now suppose we have
  - Database: subject
  - Window and Console: observer
  - Update Window and Console clients when data is modified

The Observer pattern in Java 8 (cont.)

```
// Subject.java
interface Subject {
  ArrayList<Observer> observers
    = new ArrayList<Observer>();
  default void attach(Observer o) {
    observers.add(o);
  }
  default void detach(Observer o) {
    observers.remove(o);
  }
  default void doNotify(String s) {
    observers.forEach(o -> o.update(s));
  }
}

// Observer.java
interface Observer {
  default void update(String s) {
    System.out.println(this
      + " is updated for " + s);
  }
}
```

- Let database implement Subject, Window and Console implement Observer
- So far so good

```
// Database.java
class Database implements Subject {}

// Window.java
class Window implements Observer {}

// Console.java
class Console implements Observer {}

// Test.java
public class Test {
  public static void main(String[] args) {
    Database db = new Database();
    Window w = new Window();
    Console c = new Console();
    db.attach(w);
    db.attach(c);
    db.doNotify("test");
  }
}
```

```
observer.Window@816f27d is updated for test
observer.Console@87aac27 is updated for test
```

The Observer pattern in Java 8 (cont.)

```
// Subject.java
interface Subject {
  ArrayList<Observer> observers
    = new ArrayList<Observer>();
  default void attach(Observer o) {
    observers.add(o);
  }
  default void detach(Observer o) {
    observers.remove(o);
  }
  default void doNotify(String s) {
    observers.forEach(o -> o.update(s));
  }
}

// Observer.java
interface Observer {
  default void update(String s) {
    System.out.println(this
      + " is updated for " + s);
  }
}
```

- But how could we switch on/off?
  - To use the Observer pattern or not
  - Directly modify Database, Window, and Console

```
// Database.java
class Database implements Subject {}

// Window.java
class Window implements Observer {}

// Console.java
class Console implements Observer {}

// Test.java
public class Test {
  public static void main(String[] args) {
    Database db = new Database();
    Window w = new Window();
    Console c = new Console();
    db.attach(w);
    db.attach(c);
    db.doNotify("test");
  }
}
```

```
observer.Window@816f27d is updated for test
observer.Console@87aac27 is updated for test
```

The observer pattern in AspectJ

```
aspect ObserverPattern {
  interface Subject {
    ArrayList<Observer> observers
      = new ArrayList<Observer>();
    default void attach(Observer o) {
      observers.add(o);
    }
    default void detach(Observer o) {
      observers.remove(o);
    }
    default void doNotify(String s) {
      observers.forEach(o -> o.update(s));
    }
  }
  interface Observer {
    default void update(String s) {
      System.out.println(this
        + " is updated for " + s);
    }
  }

  declare parents: Console implements Observer;
  declare parents: Window implements Observer;
  declare parents: Database implements Subject;
}
```

- With aspects all the code related to the Observer pattern can be gathered
- The two interfaces and the mapping can be put into an aspect
- Without modifying the original code!

→ Obliviousness

The observer pattern in AspectJ (cont.)

- Obliviousness
  - An important property of AOP
- It might not always be a good thing, but is helpful in a large system
- Easy to plug and unplug

Reference

- *Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and aspectJ. In the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02), 2002.*

Procedure? Function? Method?

- In some contexts they are used interchangeably
  - terminology
- A very basic modular unit
  - A piece of code for a certain purpose
  - Supported by language constructs
- For reuse, composition, maintainability, etc.

Procedure

A way of doing something, especially the usual or correct way.

--- Oxford Advanced Learner's Dictionary

- Procedure sounds more imperative
  - subroutine; contains a set of statements
  - step by step; what to do
  - usually not related to objects
  - E.g. procedures in Pascal

Function

A quantity whose value depends on the varying values of others.

--- Oxford Advanced Learner's Dictionary

- Function sounds more declarative, mathematical
  - defines as expressions; what it is
  - E.g. functions in Lisp, Haskell, etc.
- However
  - Also used as procedures: functions in FORTRAN, C
  - May used along with objects: C++, JavaScript, etc.

Method

A particular way of doing something.

--- Oxford Advanced Learner's Dictionary

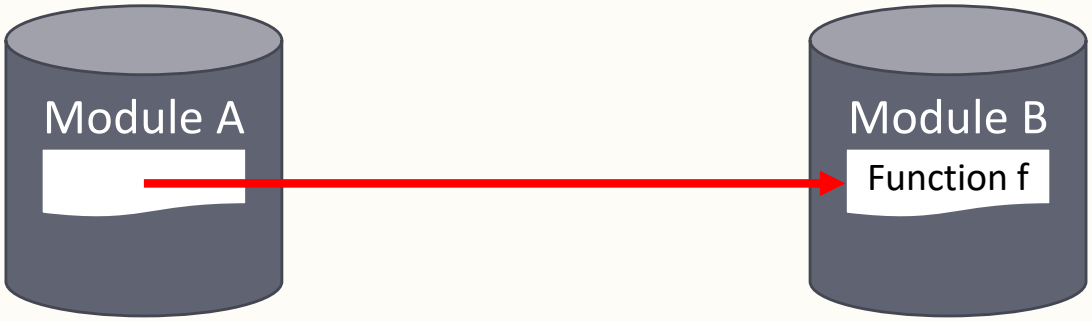
- Method implies "the functions on objects"
  - a kind of object's property; members of a class
  - response to a message
  - E.g. methods in Java

Anyway, they are kinds of constructs

- Supported by programming languages
  - allow programmers to define them
  - with keywords and signatures
- Can be invoked by calls
  - invoke and execute the contents inside them

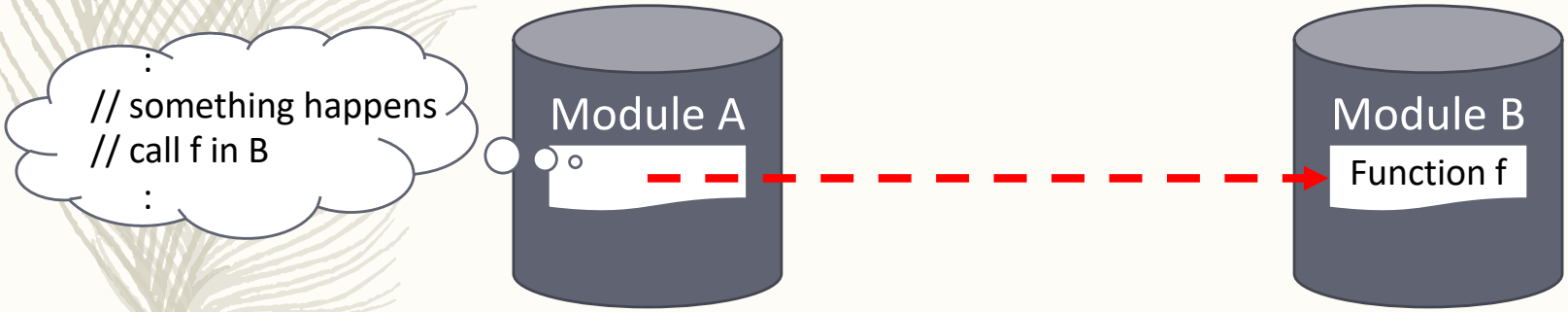
Function calls

- Immediate
  - The code inside them are executed right now
- Active
  - Call it rather than let it be called
  - E.g. in module A, calling the function f in module B



How to let a function be called later?

- Delayed
  - Execute the code when something happens
  - Passive
  - E.g. let “the function f in module B” be called by module A
  - Callback



Why we need “callback”?

– Suppose Bob want to contact Alice by telephone

“Is Alice there, please?”

“I’m sorry, Alice is not here at the moment.
Can you call again later?”

(30 minutes passed and Bob calls again)

“Is Alice there, please?”

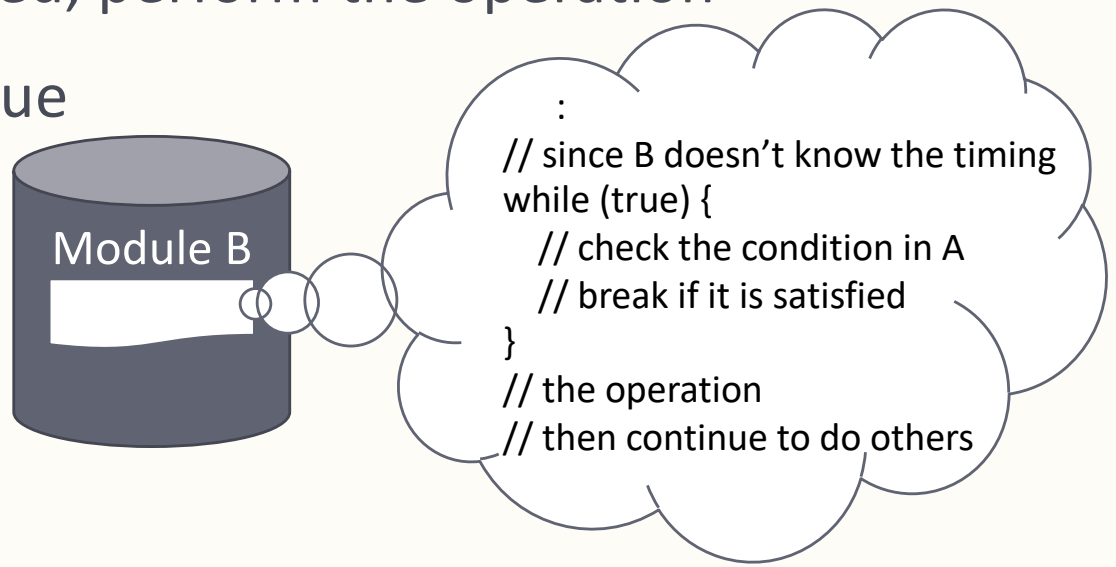
“I’m afraid she’s is stepped out.”

“Thanks, could you ask her to call Bob when she gets in?”

--- since Bob don’t want to call every 30 minutes

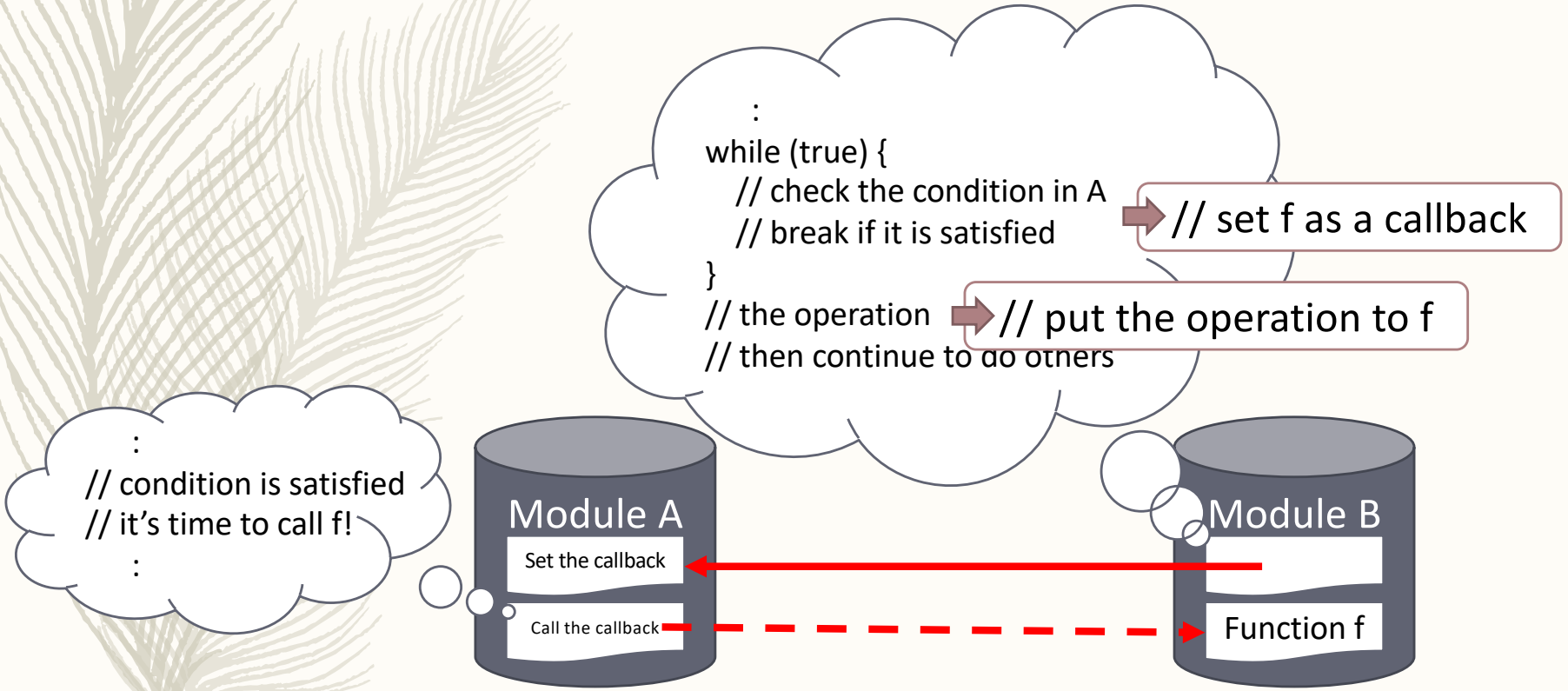
Polling: ask again and again

- In some situations we need to perform an operation after something happens
- Polling
  - Check the condition periodically
  - If it is satisfied, perform the operation
  - Then continue



Set a callback and continue

- Give the callback to the one who changes the condition
- Continue to do other operations

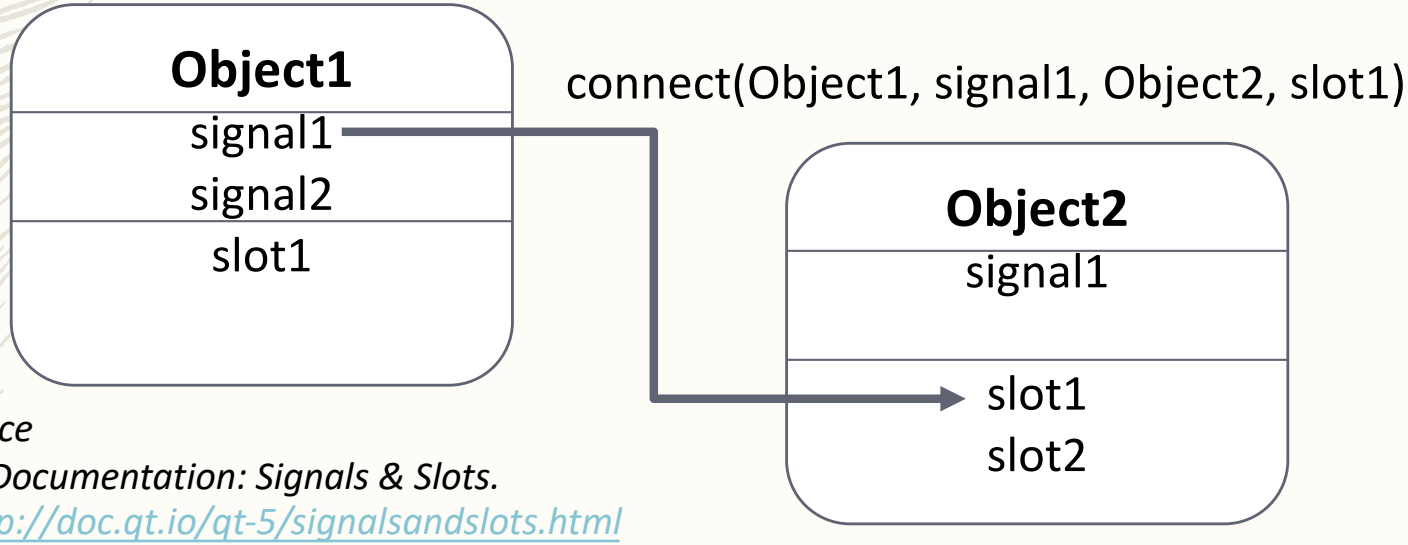


Event-driven programming

- Use events to mark happenings
  - i.e. when the condition is satisfied
- Such a callback for an event is named handler
  - the operation, the handling
- Allow to set the callback dynamically
  - instead of hard code the name of the callback
  - Loose coupling!

Examples of events

- GUI toolkits
  - Let you add a handler for a button click
  - The handler will be called when users click the button
- Not only for GUI
- E.g. the signals (events) and slots (handlers) in Qt



Reference

- *Qt Documentation: Signals & Slots.* http://doc.qt.io/qt-5/signalsandslots.html

Examples of events (cont.)

- GTK
  - Connect a callback (handler) to the signal (event) we want to handle
- Swing
  - Add an event listener to a JButton
  - Event listeners are objects that have corresponding handlers for events

Must use along with objects?

- Events are NOT necessarily used along with objects
  - But now they are widely used in OO languages
- Some languages directly support events
  - by language constructs
  - a kind of field
  - E.g. the event and delegate in C#

Events in C#

- Define the type of a handler

```
public delegate void
NotifyHandler(String s);
```

  - Such a handler takes a String and returns nothing

- Define an event

```
public event NotifyHandler Notify;
```

  - The type of handlers for this event must be NotifyHandler

- Invoke the event

```
Notify("test!");
```

Reference

- *Events (C# Programming Guide).* https://msdn.microsoft.com/en-us/library/awbftdfh.aspx

Events in C# (cont.)

```
public class Subject {
  public delegate void NotifyHandler(String s);
  public event NotifyHandler Notify;
  public void doNotify(String s) {
    if (Notify != null) {
      Notify(s);
    }
  }
}
public class Observer {
  public void Update(String s) {
    Console.WriteLine(this + " is updated for " + s + ".");
  }
  public static void Main(string[] args) {
    Subject s = new Subject();
    s.doNotify("first message");
    Observer o1 = new Observer();
    s.Notify += new Subject.NotifyHandler(o1.Update);
    s.doNotify("second message");
    Observer o2 = new Observer();
    s.Notify += new Subject.NotifyHandler(o2.Update);
    s.doNotify("third message");
    s.Notify -= new Subject.NotifyHandler(o1.Update);
    s.doNotify("fourth message");
  }
}
```

invoke the event

bind (register) the handler to the event

unbind the handler from the event

Events and handlers

- Loose the coupling between modules/objects
  - Allow to bind and unbind at runtime
  - Allow to bind multiple handlers to an event
  - Allow to bind a handler to multiple events

Sounds familiar?

→ Yes, the Observer pattern, again

The Observer pattern in Java

```
public class Subject {
  private ArrayList<Observer> observers = new ArrayList<Observer>();
  public void attach(Observer o) {
    observers.add(o);
  }
  public void detach(Observer o) {
    observers.remove(o);
  }
  public void doNotify(String s) {
    if (observers.size() > 0) {
      observers.forEach(o -> o.update(s));
    }
  }
}
```

invoke the event

```
public class Observer {
  public void update(String s) {
    System.out.println(this + " is updated for " + s + ".");
  }
  public static void main(String[] args) {
    Subject s = new Subject();
    s.doNotify("first message");
    Observer o1 = new Observer();
    s.attach(o1);
    s.doNotify("second message");
    Observer o2 = new Observer();
    s.attach(o2);
    s.doNotify("third message");
    s.detach(o1);
    s.doNotify("fourth message");
  }
}
```

bind (register) the handler to the event

unbind the handler from the event

Why we need event construct?

1. implement such code every time?

2. two observer list for two events?

3. two attach/ detach/doNotify for two events?

4. implement the code of Subject in the Observer if it also has an event?

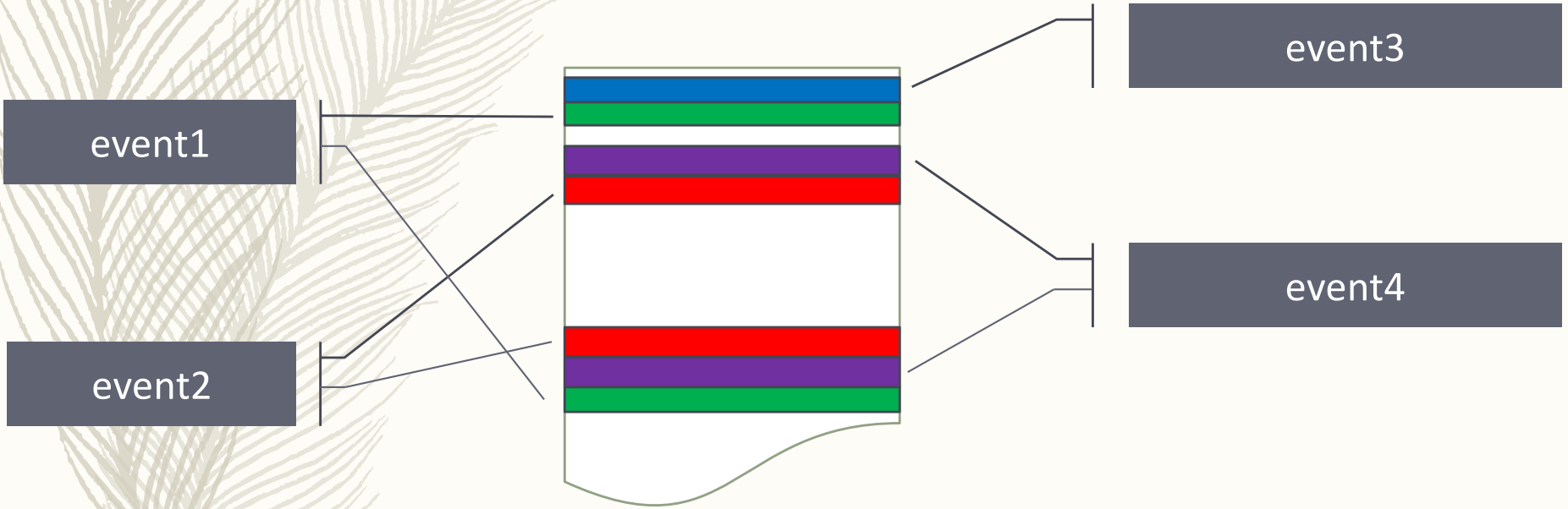
```
public class Subject {
  private ArrayList<Observer> observers = new ArrayList<Observer>();
  public void attach(Observer o) {
    observers.add(o);
  }
  public void detach(Observer o) {
    observers.remove(o);
  }
  public void doNotify(String s) {
    if (observers.size() > 0) {
      observers.forEach(o -> o.update(s));
    }
  }
}
```

```
public class Observer {
  public void update(String s) {
    System.out.println(this + " is updated for " + s + ".");
  }
  public static void main(String[] args) {
    Subject s = new Subject();
    s.doNotify("first message");
    Observer o1 = new Observer();
    s.attach(o1);
    s.doNotify("second message");
    Observer o2 = new Observer();
    s.attach(o2);
    s.doNotify("third message");
    s.detach(o1);
    s.doNotify("fourth message");
  }
}
```

– Rather than implementing the observer pattern?

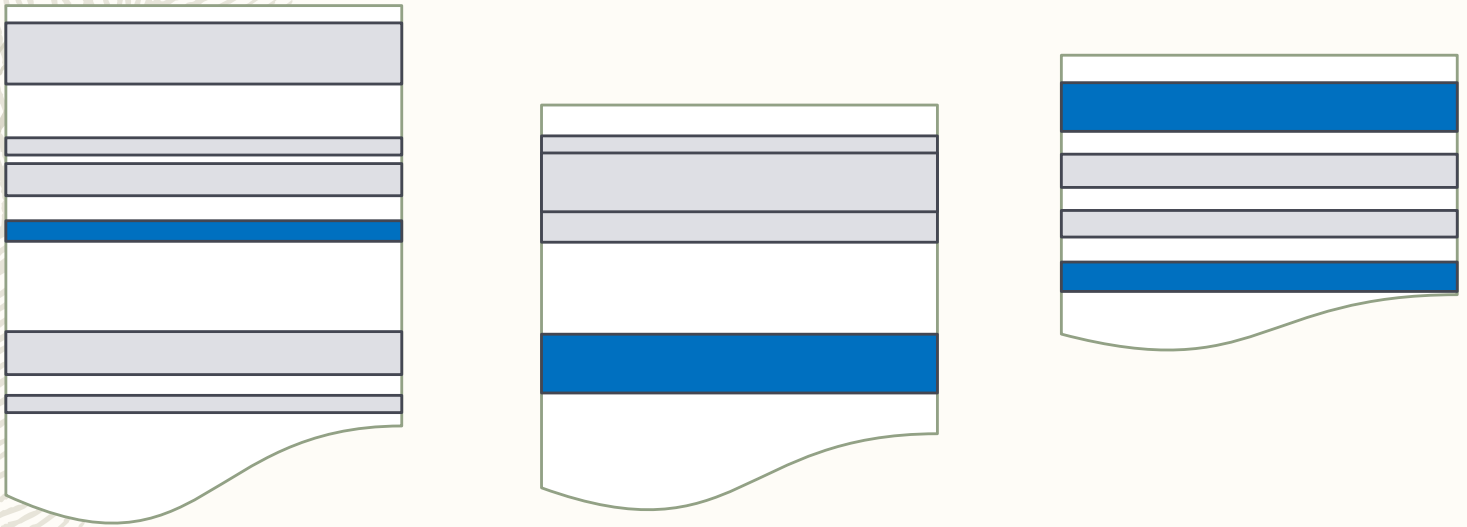
Recall: Code tangling

- The codes for different concerns are tangled
  - Not be separated well



Recall: Code scattering

- The code for one event is scattered throughout the program
  - Not be gathered in one module



Again, sounds familiar?

- Event-handler might remind you of the join point model in AOP
  - Events can be regarded as join points
    - *Event declarations are some kind of pointcuts*
  - Handlers can be regarded as after advice

- AOP and EDP are different paradigms
  - To resolve different design problems
  - But might share some language constructs!

Programming paradigms

- Until now we have seen
  - Object-Oriented Programming
  - Aspect-Oriented Programming
  - Event-Driven Programming

- They are developed to address different design problems in a software

Module compositions and coupling

- Programming paradigms help programmers to tackle modularization and composition
  - Modularize the code of a software
  - And then compose or couple them
- Now we are going to see COP

Context-oriented programming

- What is context?
  - The environment/conditions during the execution
- Why COP?
  - Enable software to dynamically adapt its behavior according to the current context at runtime
  - Dynamically compose context-dependent concerns

Reference

- *Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming — an overview of ContextL, 2005.*
- *Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming, 2008.*

Context-dependent behavior

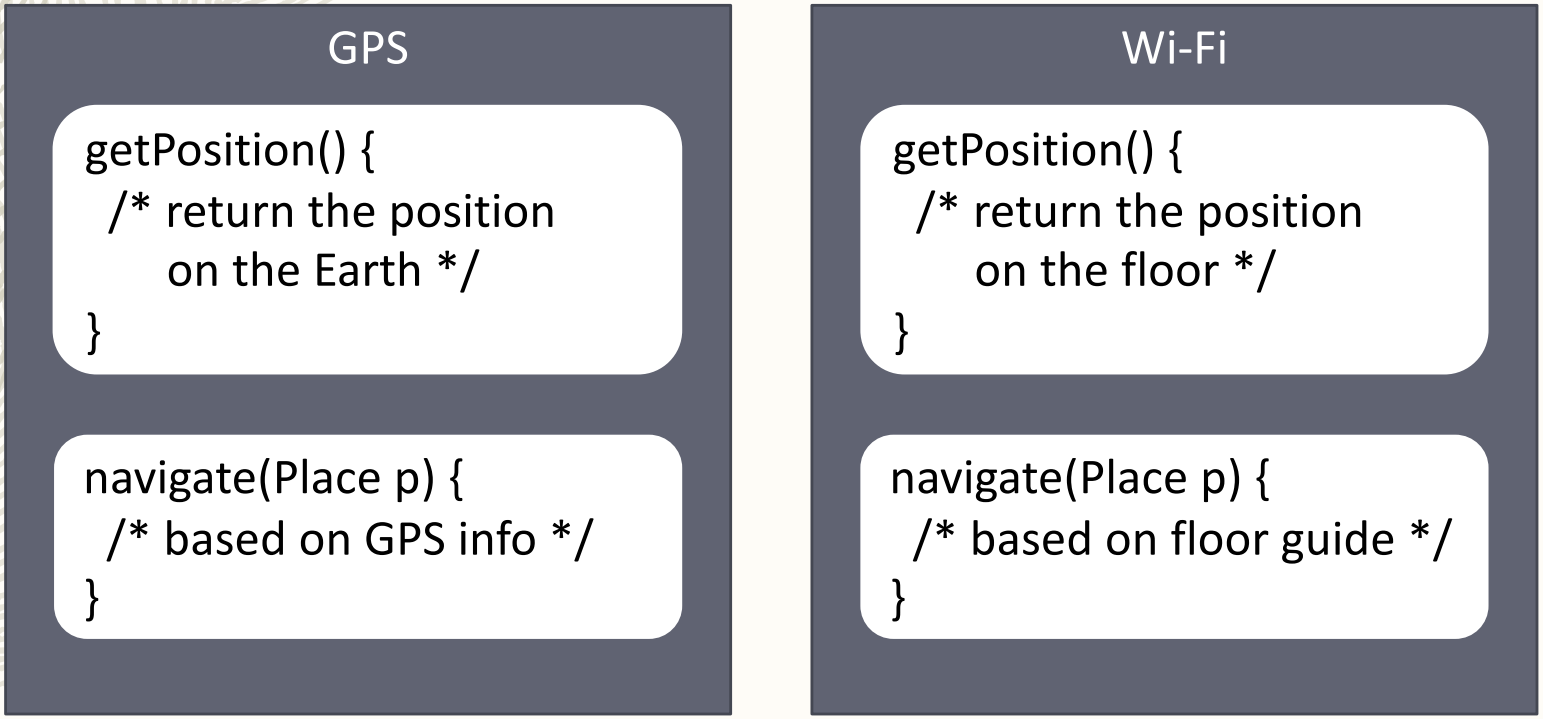
- Some behaviors in an application are dependent on current execution context
- In other words, we might need several implementations for a function
  - Invoke different implementations for a function call

Context-dependent behavior (cont.)

- For example, a mobile application that detects the position and shows it on the map
  - If GPS is available, get the position by GPS
  - When you enter a building and Wi-Fi becomes available, get the floor information from local server

Context-dependent behavior (cont.)

- For example, a mobile application that detects the position and shows it on the map
  - Depend on the context: GPS or Wi-Fi

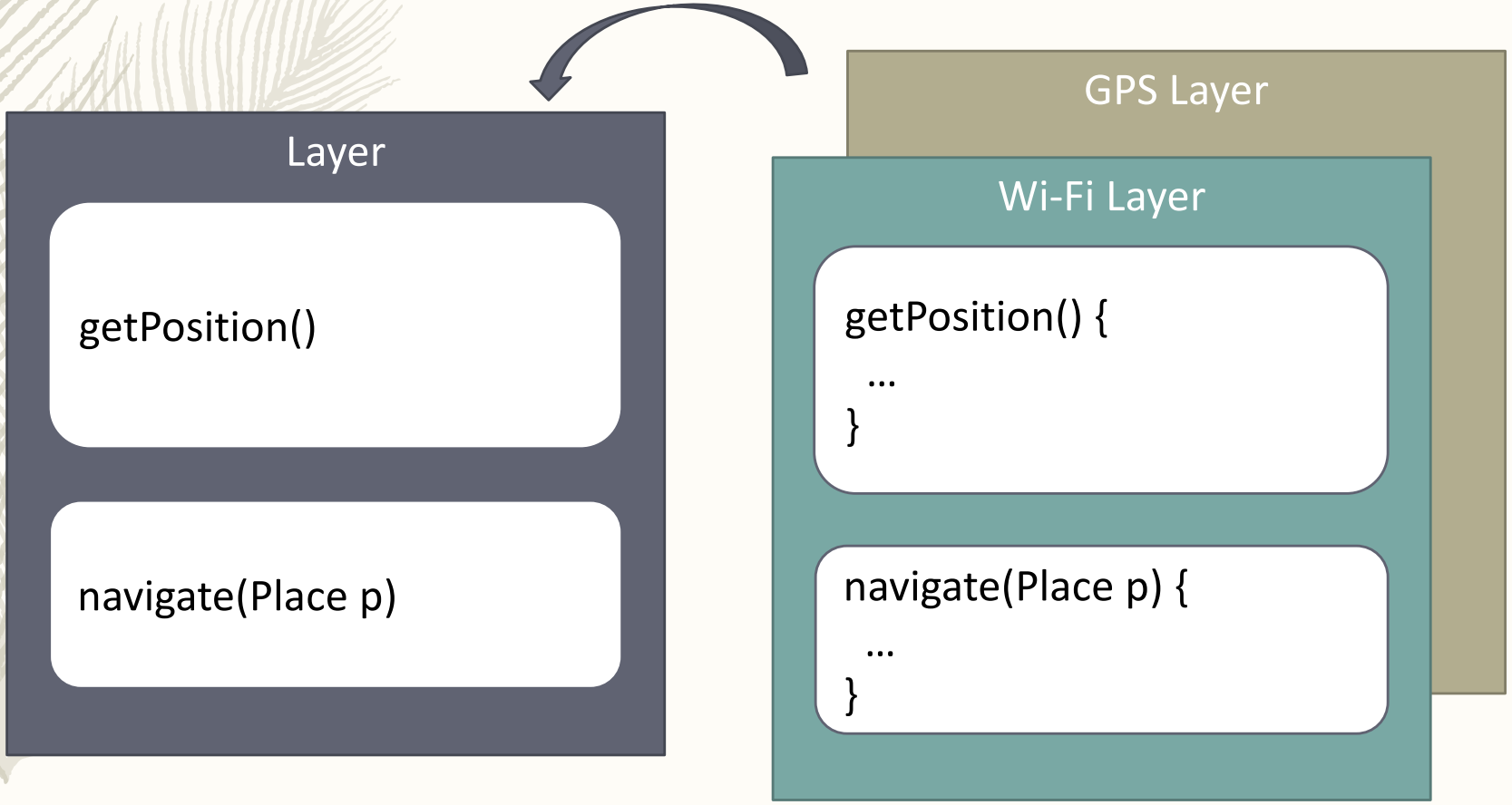


Layers: a module in CO languages

- Layer is a module for context-dependent behaviors
  - Supported by constructs in CO languages
  - Some sort of classes
- Put context-dependent behaviors in them
  - Then activate them to use

Switch behaviors by layer activation

- Modularize function implementations in a layer
  - and switch between them by activating layers



Layers represent the context

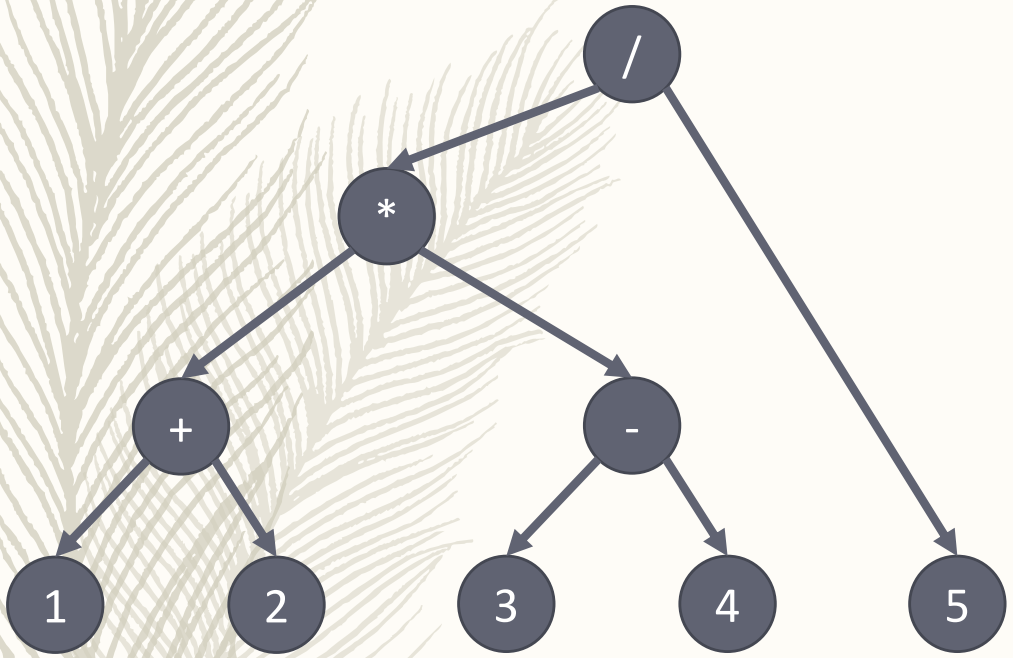
- For example, activate GPS layer when GPS is available
  - function calls to getPosition will invoke the implementation in GPS layer

```
with GPS {
        :
    p = getPosition();  // the implementation in GPS is invoked
        :
}
```

  - Similarly, in the case of Wi-Fi

```
with WiFi {
        :
    p = getPosition();  // the implementation in Wi-Fi is invoked
}
```

Another example: traversal



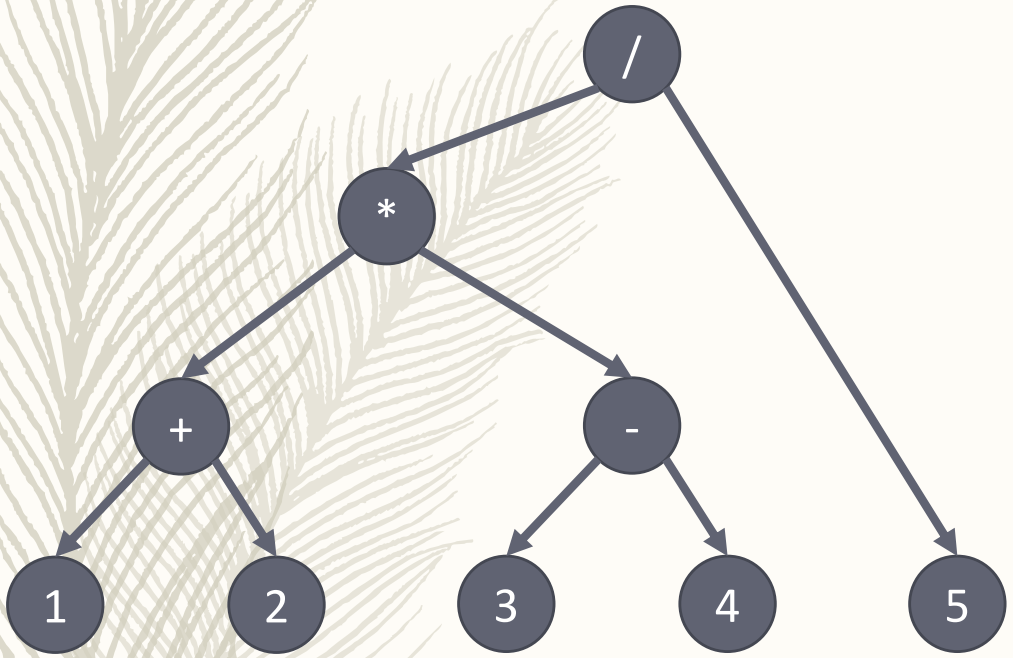
- Suppose we want to traverse a tree to
  - Print in infix notation (((1+2)*(3-4))/5)
  - Print in prefix notation (/(*(+12)(-34))5)
  - Print in postfix notation (((12+)(34-)*)5/)
  - Evaluate its value (-3/5)
- The traversal is the same but the operations are different

Reference

- *Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An Introduction to Context-Oriented Programming with ContextS, 2007.*

OO style, again

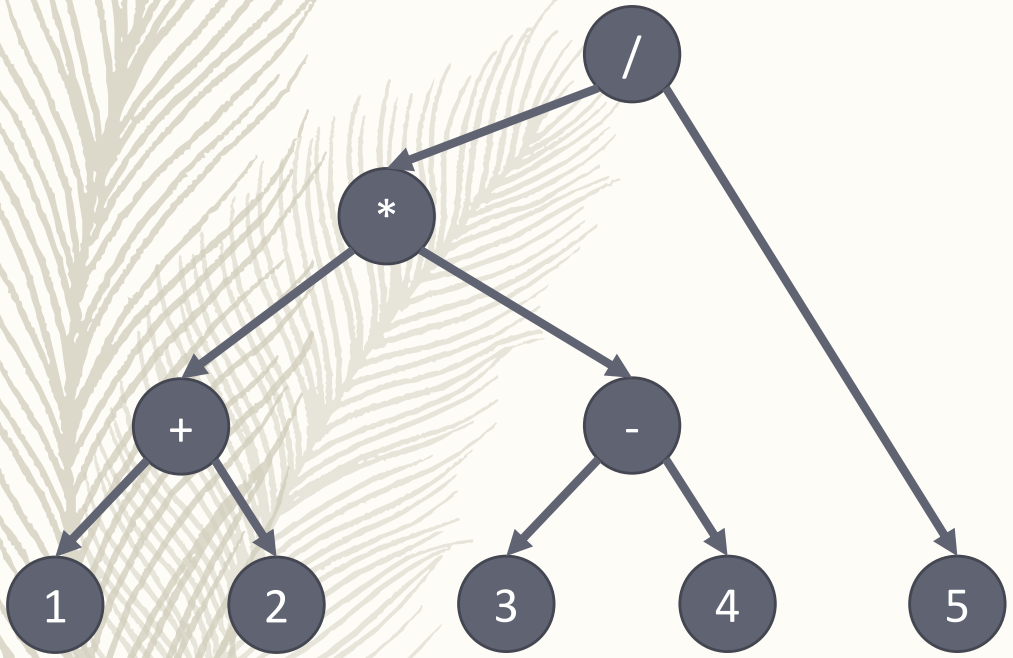
– Implement with different methods in a node




```
class Node {
  void printInfix() {
    if (isLeaf())    {
      print(get());
    } else {
      print("(");
      left().printInfix();
      print(get());
      right().printInfix();
      print(")");
    }
  }
  void printPrefix() { … }
  void printPostfix() { … }
  void eval() { … }
}
```

The Visitor pattern, again

– Different methods are defined in different visitors

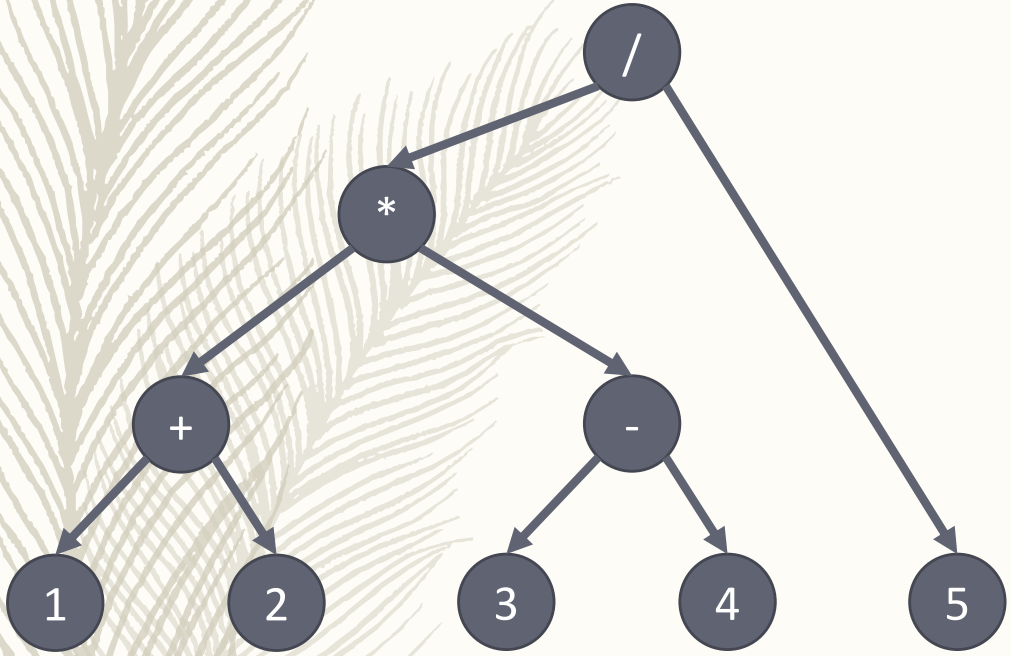



```
class InfixPrinter {
  void visit(Node n) {
    if (n.isLeaf())    {
      print(n.get());
    } else {
      print("(");
      n.left().printInfix();
      print(n.get());
      n.right().printInfix();
      print(")");
    }
  }
}
```

COP: aware of execution context

– Different methods are defined in different layers

– The implementation in the activated layer is used




```
layer InfixPrint {
  traversal() {
      :
  }
}

with InfixPrint {
    :
  traversal();
    :
}
```

Layer activation can be nested

- Search the implementation for a method from inner layers to outer layers
  - The implementation in the inner one is used
- For example,

```
with A {
   // the one in A is used
   with B {
      // the one in B is used
      with C {
          // the one in C is used
      }
      // the one in B is used
   }
   // the one in A is used
}
```

The way to activate a layer

- The **with** we have seen has a block scope
  - Nested layer activations
  - Managed by a stack
  - Lexical and simple
- Another dynamic way is possible as well
  - Some CO languages support both the two kinds of layer activation

Activate a layer “dynamically”

- We may also separate the activate and deactivate
- The implementation in the most recent one is used

```
activate A;
      :  // the one in A is used
activate B;
      :  // the one in B is used
deactivate A;
      :  // the one in B is used
deactivate B;
```

Deactivate a layer in a lexical way

- Such a design is also possible
  - Allow to deactivate a layer in the nested scope
- For example,

```
with A {
   // the one in A is used
   with B {
      // the one in B is used
      without B {
         // the one in A is used
      }
      // the one in B is used
   }
   // the one in A is used
}
```

Layer compositions and refinements

- Several CO languages allow to
  - Compose layers, i.e. mixin
    - *mix A with B*
  - Refine a layer, i.e. subclassing
    - *A inherit from B*
- Use along with or without classes

A kind of Aspect-Oriented Programming?

- COP can be regarded as some sort of AOP
  - The aspect is context!
  - Context-dependent concerns are potentially crosscutting
- But what COP focuses on is dynamic activation
  - On the other hand, obliviousness and quantification are important in AOP

Call the “previous” implementation?

- In OOP, we may override a method
  - and use super to call the overridden one
- In AOP, we may use around advice to replace the original implementation
  - and use proceed to call the original one
- In COP, we may activate a layer to use a different implementation
  - and have some way to call the previous one

Next lecture

12/23 **Practice 7**

12/30 Generic Programming and Metaprogramming