7035 Programmierung 2

Object-Oriented Programming with Java

Prof. O. Nierstrasz

Sommersemester 2000

Table of Contents

Patterns, Rules and Guidelines

1. P2 — Object-Oriented Programming

Lecturer:	Prof. Oscar Nierstrasz Schützenmattstr. 14/103, Tel: 631.4618, Email: oscar@iam.unibe.ch
Secretary:	Frau I. Huber, Tel. 631.4692
Assistants:	Sander Tichelaar, Frank Buchli, Marc Hugi, Daniel Tschan
WWW:	http://www.iam.unibe.ch/~scg/Teaching/P2/ *(includes full examples)*

Principle Texts:

- James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language Reference Manual,* Addison-Wesley, 1999
- David Flanagan, *Java in Nutshell: 2nd edition*, O'Reilly, 1997.
- Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1997.
- Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

Overview

1.	31.03	Introduction
2.	07.04	Design by Contract
3.	14.04	Testing and Debugging
	21.04	*Good Friday*
4.	28.04	Iterative Development
5.	05.05	Inheritance and Refactoring
6.	12.05	Programming Tools
7.	19.05	A Testing Framework
8.	26.05	Collections
9.	02.06	GUI Construction
10.	09.06	Clients and Servers
11.	16.06	Guidelines, Idioms and Patterns
12.	23.06	Common Errors, a few Puzzles
	30.06	*Final Exam*

Goals of this course

Object-Oriented Design

- ❑ How to use *responsibility-driven design* to split systems into objects
- ❑ How to exploit inheritance to make systems *generic* and *flexible*
- ❑ How to *iteratively refactor* systems to arrive at simple, clean designs

Software Quality

- ❑ How to use *design by contract* to develop robust software
- ❑ How to *test* and *validate* software

Communication

- ❑ How to keep software as *simple* as possible
- ❑ How to write software that *communicates* its design
- ❑ How to *document* a design

Skills, Techniques and Tools

- ❑ How to use debuggers, version control systems, profilers and other tools
- ❑ How and when to use standard software *components* and *architectures*
- ❑ How and when to apply common *patterns, guidelines* and *rules of thumb*

What is programming?

- ❑ Implementing data structures and algorithms?
- ❑ Writing instructions for machines?
- ❑ Implementing client specifications?
- ❑ Coding and debugging?
- ❑ Plugging together software components?
- ❑ Specification? Design?
- ❑ Testing?
- ❑ Maintenance?

Which of these are “not programming”?

Programming and Software Development

- ❑ How do you get your requirements?
- ❑ How do you know that the documented requirements reflect the user’s needs?
- ❑ How do you decide what priority to give each requirement?
- ❑ How do you select a suitable software architecture?
- ❑ How do you do detailed design?
- ❑ How do you know your implementation is “correct”?
- ❑ How, when and what do you test?
- ❑ How do you accommodate changes in requirements?
- ❑ How do you know when you’re done?

Is “programming” distinct from “software development”?

Programming activities

- ❑ Documentation
- ❑ Prototyping
- ❑ Interface specification
- ❑ Integration
- ❑ Reviewing
- ❑ Refactoring
- ❑ Testing
- ❑ Debugging
- ❑ Profiling
- ❑ ...

What do these activities have in common?

What is a software system?

A *computer program* is an application that solves a single task:

- ❑ requirements are typically well-defined
- ❑ often single-user at a time
- ❑ little or no configuration required

A *software system* is an application that supports multiple tasks.

- ❑ open requirements
- ❑ multiple users
- ❑ implemented by a set of programs or modules
- ❑ multiple installations and configurations
- ❑ long-lived (never “finished”)

Software systems are fundamentally more complex than simple programs.
Programming techniques address systems development by *reducing complexity.*

What is good (bad) design?

Consider two programs with *identical behaviour*.

- ❏ Could the one be well-designed and the other badly-designed?
- ❏ What would this mean?

A procedural design

How can we compute the total area of a set of geometric shapes?

A typical, procedural solution:

```
public static long sumShapes(Shape shapes[]) {
  long sum = 0;
  for (int i=0; i<shapes.length; i++) {
    switch (shapes[i].kind()) {
    case Shape.RECTANGLE:              // a class constant
      sum += shapes[i].rectangleArea();
      break;
    case Shape.CIRCLE:
      sum += shapes[i].circleArea();
      break;
    ... // more cases
    }
  }
  return sum;
}
```

An object-oriented approach

A typical object-oriented solution:

```
public static long sumShapes(Shape shapes[]) {
   long sum = 0;
   for (int i=0; i<shapes.length; i++) {
      sum += shapes[i].area();
   }
   return sum;
}
```

What are the advantages and disadvantages of the two solutions?

Object-Oriented Design

OO vs. functional design ...

> *Object-oriented [design] is the method which bases the architecture of any software system on the objects it manipulates (rather than “the” function it is meant to ensure).*
>
> *Ask not first what the system does: ask what it does it to!*
>
> *— Meyer, OOSC*

Responsibility-Driven Design

RDD factors a software system into objects with well-defined *responsibilities*:

- ❑ Objects are responsible to *maintain information* and *provide services:*
  - ☞ Operations are always associated to responsible objects
  - ☞ Always *delegate* to another object what you cannot do yourself
- ❑ A good design exhibits:
  - ☞ *high cohesion* of operations and data within classes
  - ☞ *low coupling* between classes and subsystems
- ❑ Every method should perform one, well-defined task:
  - ☞ *Separation of concerns* — reduce complexity
  - ☞ High level of abstraction — *write to an interface, not an implementation*
- ❑ Iterative Development
  - ☞ *Refactor* the design as it evolves

Refactoring

Refactor your design whenever the code starts to hurt:

- ❑ methods that are too long or hard to read
  - ☞ decompose and delegate responsibilities
- ❑ duplicated code
  - ☞ factor out the common parts (template methods etc.)
- ❑ violation of encapsulation, or
- ❑ too much communication between objects (high coupling)
  - ☞ reassign responsibilities
- ❑ big case statements
  - ☞ introduce subclass responsibilities
- ❑ hard to adapt to different contexts
  - ☞ separate mechanism from policy

...

What is Software Quality?

Correctness is the ability of software products to perform their exact tasks, as defined by their specifications

Robustness is the ability of software systems to react appropriately to abnormal conditions

Extendibility is the ease of adapting software products to changes of specification

Reusability is the ability of software elements to serve for the construction of many different applications

Compatibility is the ease of combining software elements with others

Efficiency is the ability of a software system to place as few demands as possible on hardware resources

Portability is the ease of transferring software products to various hardware and software environments

Ease of use is the ease with which people of various backgrounds and qualifications can learn to use software products

— *Meyer, OOSC, ch. 1*

How to achieve software quality

Design by Contract

- ❑ Pre- and post-conditions, Class invariants
- ❑ Disciplined exceptions

Standards

- ❑ Protocols, interfaces, components, libraries, frameworks
- ❑ Software architectures, design patterns

Testing and Debugging

- ❑ Unit tests, system tests ...
- ❑ Repeatable *regression tests*

Do it, do it right, do it fast

- ❑ Aim for *simplicity* and *clarity*, not performance
- ❑ Fine-tune performance only when there is a *demonstrated need!*

What is a programming language?

A programming language is a tool for:

- ❑ specifying instructions for a computer
- ❑ expressing data structures and algorithms
- ❑ communicating a design to another programmer
- ❑ describing software systems at various levels of abstraction
- ❑ specifying configurations of software components

A programming language is a tool for communication!

Communication

How do you write code that communicates its design?

- ❑ Do the simplest thing you can think of (KISS)
  - ☞ Don't over-design
  - ☞ Implement things once and only once
- ❑ Program so your code is (largely) self-documenting
  - ☞ Write small methods
  - ☞ Say what you want to do, not how to do it
- ❑ Practice reading and using other people’s code
  - ☞ Subject your code to reviews

Why use object-oriented programming?

- ❑ Modelling
  - ☞ complex systems can be naturally decomposed into software objects
- ❑ Data abstraction
  - ☞ clients are protected from variations in implementation
- ❑ Polymorphism
  - ☞ clients can uniformly manipulate plug-compatible objects
- ❑ Component reuse
  - ☞ client/supplier contracts can be made explicit, simplifying reuse
- ❑ Evolution
  - ☞ classes and inheritance limit the impact of changes

Why Java?

Special characteristics

❑ Resembles C++ *minus the complexity*

❑ *Clean integration* of many features

❑ *Dynamically loaded* classes

❑ Large, *standard* class library

Simple Object Model

❑ “Almost everything is an object”

❑ No pointers

❑ Garbage collection

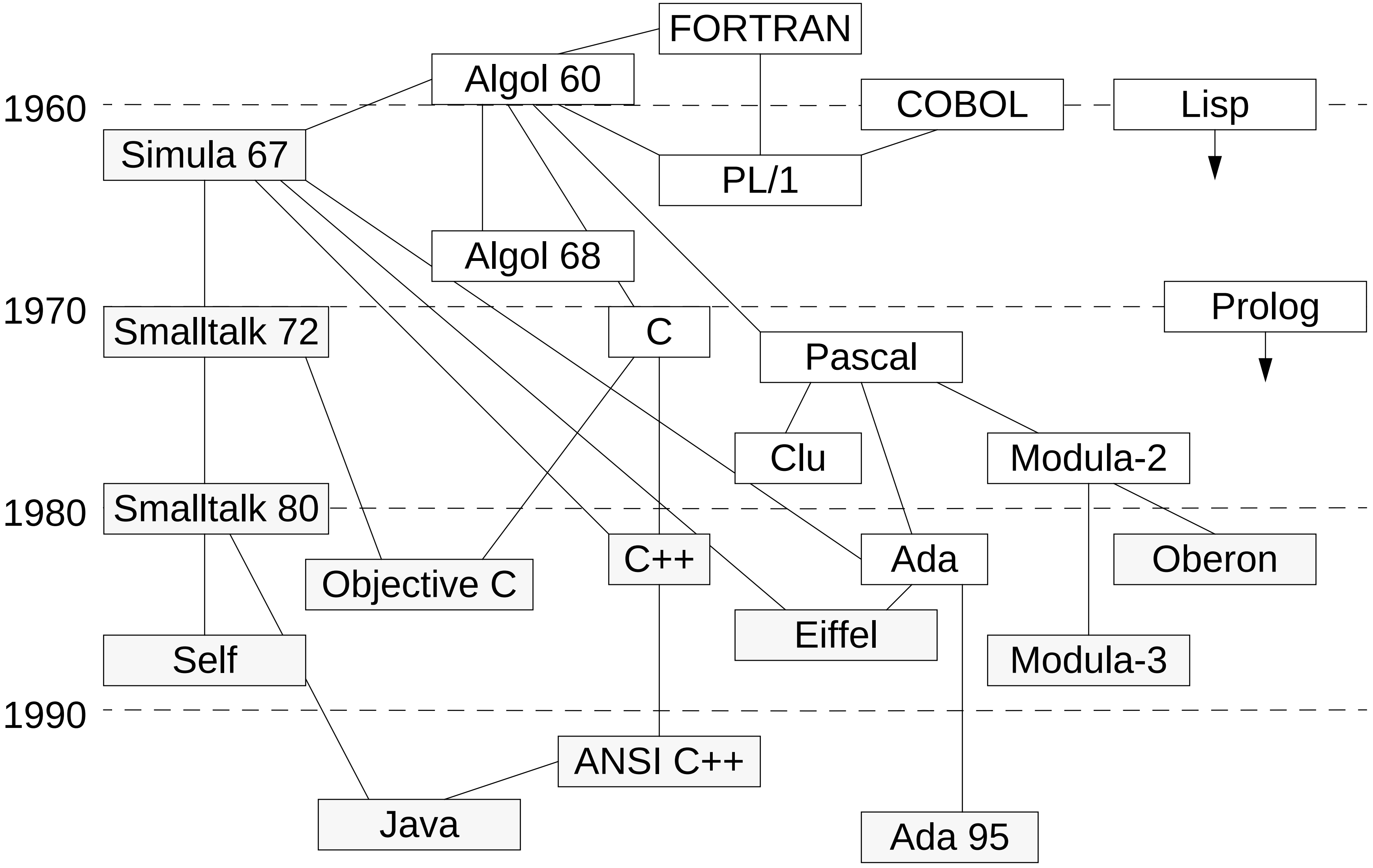
❑ Single inheritance

❑ Multiple subtyping

❑ Static *and* dynamic type-checking

Few innovations, but reasonably clean, simple and usable.

History


FORTRAN
Algol 60
COBOL
Lisp
1960
Simula 67
PL/1
Algol 68
1970
Smalltalk 72
C
Prolog
Pascal
Clu
Modula-2
1980
Smalltalk 80
C++
Ada
Oberon
Objective C
Eiffel
Self
Modula-3
1990
ANSI C++
Java
Ada 95

Summary

You should know the answers to these questions:

- ❑ What is the difference between a computer program and a software system?
- ❑ What defines a good object-oriented design?
- ❑ When does software need to be refactored? Why?
- ❑ What is “software quality”?
- ❑ How does OOP attempt to ensure high software quality?

Can you answer the following questions?

- ✎ *What does it mean to “violate encapsulation”? Why is that bad?*
- ✎ *Why shouldn’t you try to design your software to be efficient from the start?*
- ✎ *When might it be “all right” to duplicate code?*
- ✎ *How do you program classes so they will be “reusable”? Are you sure?*

2. Design by Contract

Overview

❑ Declarative programming and Data Abstraction

❑ Abstract Data Types

❑ Class Invariants

❑ Programming by Contract: pre- and post-conditions

❑ Assertions and Disciplined Exceptions

Source

❑ Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1997.

Stacks

A *Stack* is a classical data abstraction with many applications in computer programming.

A Stack supports (at least) two mutating operations (`push` and `pop`) and one querying operation (`top`).

Operation	*Stack*	*isEmpty()*	*size()*	*top()*
		true	0	(error)
push(6)	6	false	1	6
push(7)	6 7	false	2	7
push(3)	6 7 3	false	3	3
pop()	6 7	false	2	7
push(2)	6 7 2	false	3	2
pop()	6 7	false	2	7

Example: Balancing Parentheses

Problem:

Determine whether an expression containing parentheses ( ), brackets [ ] and braces { } is correctly balanced.

Examples:

"`if (a.b()) { c[d].e(); } else { f[g][h].i(); }`" is balanced,
"`((a+b())`" is not balanced.

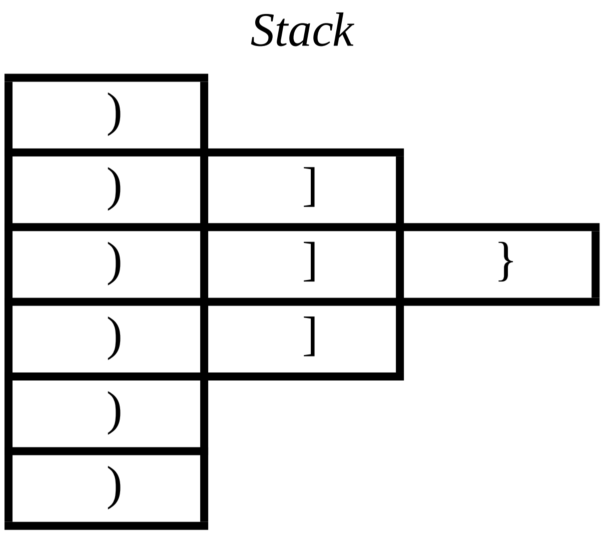
Approach:

- ❑ when you read a left parenthesis, push the matching parenthesis on a stack
- ❑ when you read a right parenthesis, compare it to the value on top of the stack
  - ☞ if they match, you pop and continue
  - ☞ if they mismatch, the expression is not balanced
- ❑ if the stack is empty at the end, the whole expression is balanced, otherwise not

Using a Stack to match parentheses

Sample input: “([{ }]]”

Input	*Case*	*Op*
(	left	push)
[	left	push]
{	left	push }
}	match	pop
]	match	pop
]	mismatch	^false



The ParenMatch class

A ParenMatch object uses a stack to check if parentheses in a text String are balanced:

```
public class ParenMatch {
  String _line;
  StackInterface _stack;
  public ParenMatch(String line, StackInterface stack) {
    _line = line; _stack = stack;
  }
  public String reportMatch() throws AssertionException {
    if (_line == null) { return ""; }
    return "\"" + _line + "\" is"
      + (this.parenMatch() ? " " : " not ")
      + "balanced";
  }
  ...
}
```

A declarative algorithm

We implement our algorithm at the highest level of abstraction possible.

```
public boolean parenMatch() throws AssertionException {
  for (int i=0; i<_line.length(); i++) {
    char c = _line.charAt(i);
    if (isLeftParen(c)) {      // expect right paren later
      _stack.push(new Character(matchingRightParen(c)));
    } else {
      if (isRightParen(c)) {   // should be on top of stack!
        if (_stack.isEmpty()) { return false; }
        if (_stack.top().equals(new Character(c))) {
          _stack.pop();        // ok, so continue
        } else { return false; } // not ok
      }
    }
  }
  return _stack.isEmpty();     // no missing right parens?
}
```

Helper methods

The helper methods are trivial to implement, and their details only get in the way of the main algorithm.

```
private boolean isLeftParen(char c) {
  return (c == '(') || (c == '[') || (c == '{');
}

private boolean isRightParen(char c) {
  return (c == ')') || (c == ']') || (c == '}');
}

private char matchingRightParen(char c) {
  switch (c) {
    case '(' : return ')';
    case '[' : return ']';
    case '{' : return '}';
  }
  return c;
}
```

What is Data Abstraction?

An *implementation* of a stack consists of:

❑ a *data structure* to *represent the state* of the stack

❑ a set of *operations* that *access* and *modify* the stack

Encapsulation means *bundling together related entities.*

Information hiding means *exposing an abstract interface and hiding the rest.*

An Abstract Data Type (ADT):

❑ *encapsulates* data and operations, and

❑ *hides* the implementation behind a well-defined interface.

StackInterface

There are many ways to implement stacks. Let us first specify an interface:

```
public interface StackInterface {
  public boolean isEmpty();
  public int size();
  public void push(Object item) throws AssertionException;
  public Object top() throws AssertionException;
  public void pop() throws AssertionException;
}
```

The methods that might fail are declared to throw an AssertionException.

➤ How do you let clients respond to multiple implementations of an ADT?

✔ *Specify an interface or an abstract class.*

Interfaces in Java

Interfaces reduce *coupling* between objects and their clients:

- ❑ A class can *implement* multiple interfaces
  - ☞ ... but can only *extend* one parent class
- ❑ Clients should *depend on an interface*, not an implementation
  - ☞ ... so implementations don’t need to extend a specific class

Define an interface for any ADT that will have more than one implementation

Exceptions

All Exception classes look like this!

You define your own exception class to distinguish your exceptions from any other kind.

The implementation consists of a default constructor, and a constructor that takes a simple message string as an argument. Both constructors call super() to ensure that the instance is properly initialized.

```
public class AssertionException extends Exception {
  AssertionException() { super(); }
  AssertionException(String s) { super(s); }
}
```

Why are ADTs important?

Communication

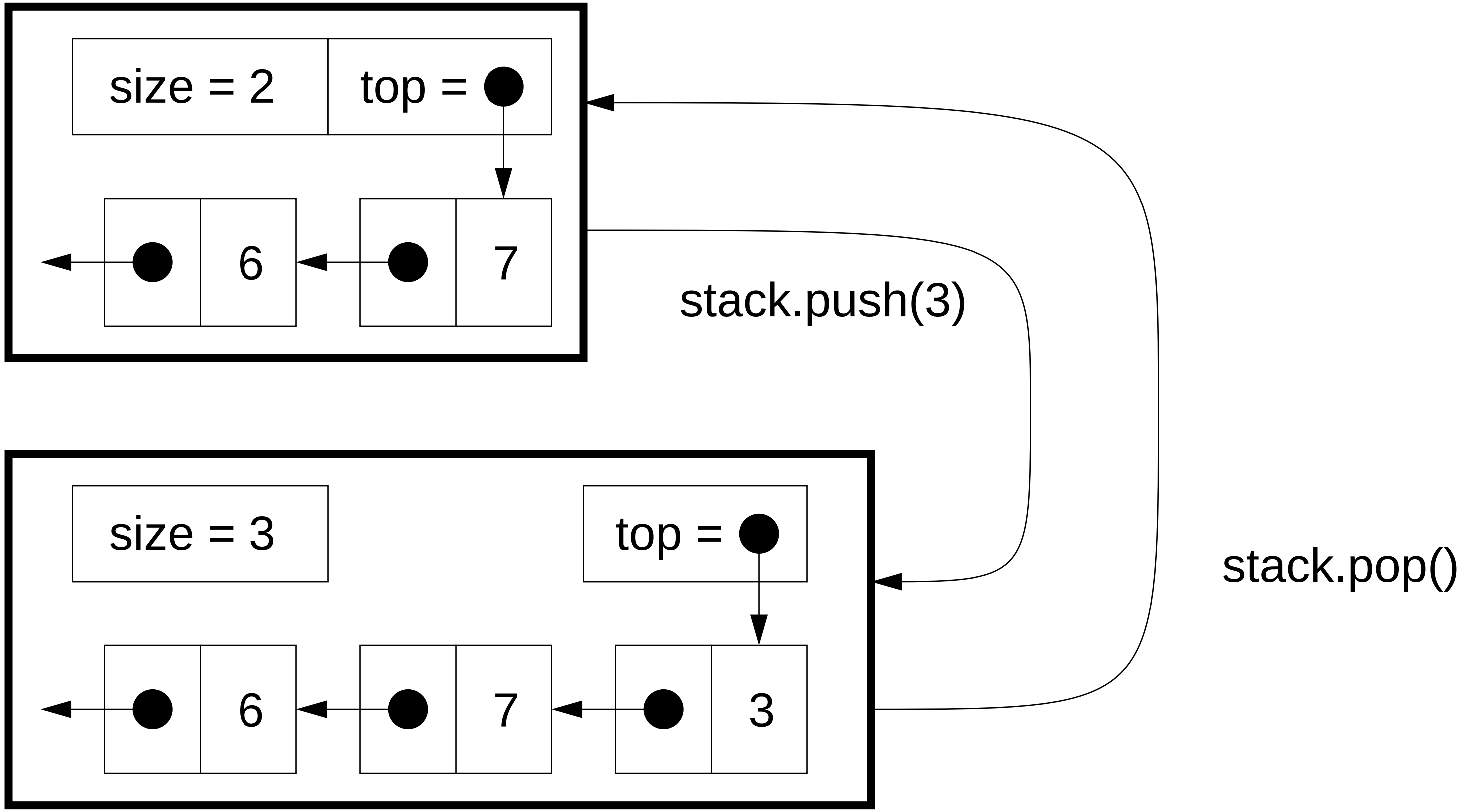
- ❑ An ADT exports *what a client needs to know*, and nothing more!
- ❑ By using ADTs, you communicate *what you want to do*, not how to do it!
- ❑ ADTs allow you to *directly model your problem domain* rather than how you will use to the computer to do so.

Software Quality and Evolution

- ❑ ADTs help to *decompose a system into manageable parts*, each of which can be separately implemented and validated.
- ❑ ADTs *protect clients from changes* in implementation.
- ❑ ADTs encapsulate client/server *contracts*
- ❑ *Interfaces* to ADTs *can be extended* without affecting clients.
- ❑ *New implementations* of ADTs can be transparently *added* to a system.

Stacks as Linked Lists

A Stack can easily be implemented using a linked data structure:



LinkClass Cells

We can define the Cells of the linked list as an *inner class* within LinkStack:

```
public class LinkStack implements StackInterface { ...
  public class Cell {
    public Object item;
    public Cell next;
    public Cell(Object item, Cell next) {
      this.item = item;
      this.next = next;
    }
  } ...
}
```

➤ When should instance variables be public?

✔ *Always make instance variables private or protected.*

The Cell class is a special case, since its instances are strictly private to LinkStack.

LinkStack ADT

```
public class LinkStack implements StackInterface {
  private Cell _top;
  private int _size;

  public LinkStack() {
    // Establishes the invariant.
    _top = null;
    _size = 0;
  }
```

➤ How should you name a private or protected instance variable?

✔ *Pick a name that reflects the role of the variable.*

✔ *Tag the name with an underscore (_).*

Role-based names tell the reader of a class what the purpose of the variables is.
A tagged name reminds the reader that a variable represents hidden state.

Class Invariants

A class invariant is any condition that expresses the *valid states* for objects of that class:

- ❑ it must be *established* by every constructor
- ❑ every public method
  - ☞ may *assume* it holds when the method starts
  - ☞ must *re-establish* it when it finishes

Stack instances must satisfy the following invariant:

- ❑ size ≥ 0

LinkStack Class Invariant

A valid LinkStack instance has a integer `_size`, and a `_top` that points to a sequence of linked Cells, such that:

- ❑ `_size` is always ≥ 0
- ❑ When `_size` is zero, `_top` points nowhere (`== null`)
- ❑ When `_size > 0`, `_top` points to a `Cell` containing the top item

Programming by Contract

Every ADT is designed to provide certain *services* given certain *assumptions* hold.

An ADT establishes a contract with its clients by associated a *precondition* and a *postcondition* to every operation O, which states:

> “If you promise to call O with the *precondition* satisfied, then I, in return, promise to deliver a final state in which the *postcondition* is satisfied.”

Consequence:

- ❑ if the precondition does not hold, the ADT is *not required to provide anything!*

Pre- and Postconditions

The *precondition* binds *clients:*

- ❑ it defines what the ADT *requires* for a call to the operation to be legitimate.
- ❑ it may involve initial state and arguments.

The *postcondition*, in return, binds the *supplier:*

- ❑ it defines the conditions that the ADT *ensures* on return.
- ❑ it may only involve the initial and final states, the arguments and the result

	Obligations	*Benefits*
Client	Only call `pop()` on a non-empty stack	Stack `size` decreases by 1. Top element is removed.
Supplier	Decrement the `size`. Remove the top element.	No need to handle case when stack is empty.

Stack pre- and postconditions

isEmpty()

- requires: - *always valid*
- ensures: - *no state change*

size()

- requires: - *always valid*
- ensures: - *no state change*

push(Object item)

- requires: - *always valid*
- ensures: not empty, size == old size + 1, top == item

top()

- requires: not empty
- ensures: - *no state change*

pop()

- requires: not empty
- ensures: size == old size -1

Assertions

An assertion is any boolean expression we expect to be true at some point :

Assertions have four principle applications:

1. Help in writing correct software
   - ☞ formalizing invariants, and pre- and post-conditions
2. Documentation aid
   - ☞ specifying contracts
3. Debugging tool
   - ☞ testing assertions at run-time
4. Support for software fault tolerance
   - ☞ detecting and handling failures at run-time

➤ What should an object do if an assertion does not hold?

✔ *Throw an exception.*

Testing Assertions

It is easy to add an assertion-checker to a class:

(unfortunately this method is not defined in java.lang.Object)

```
private void assert(boolean assertion)
  throws AssertionException {
  if (!assertion) {
    throw new AssertionException(
      "Assertion failed in LinkStack");
  }
}
```

Every class will have its own invariant:

```
private boolean invariant() {
  return (_size >= 0) &&
    ((_size == 0 && this._top == null)
    || (_size > 0 && this._top != null));
}
```

Disciplined Exceptions

An exception is the occurrence of an abnormal condition during the execution of a software element.

A failure is the inability of a software element to satisfy its purpose.

An error is the presence in the software of some element not satisfying its specification.

There are only two reasonable ways to react to an exception:

1. clean up the environment and report *failure* to the client ("organized panic")
2. attempt to change the conditions that led to failure and *retry*

It is not acceptable to return control to the client without special notification.

➤ When should an object throw an exception?

✔ *If and only if an assertion is violated*

If it is not possible to run your program without raising an exception, then you are abusing the exception-handling mechanism!

LinkStack methods

```
public boolean isEmpty() { return this.size() == 0; }
public int size() { return _size; }

public Object top() throws AssertionException {
  assert(!this.isEmpty()); // pre-condition
  return _top.item;
}
```

➤ Which assertions should you check?

✔ *Always check pre-conditions to methods.*

✔ *Check post-conditions and invariants if the implementation is non-trivial.*

Asserting pre-conditions lets you inform clients when *they* violate the contract.
Asserting post-conditions and invariants lets you know when *you* violate the contract.

Push and Pop

```
public void pop() throws AssertionException {
   assert(!this.isEmpty());     // pre-condition
   _top = _top.next;
   _size--;
   assert(invariant());         // NB: state change
}

public void push(Object item) throws AssertionException {
   _top = new Cell(item, _top);
   _size++;
   assert(!this.isEmpty());     // post-condition
   assert(this.top() == item);  // post-condition
   assert(invariant());
}
```

Explicit post-conditions and invariants make it easier to debug the implementation.

Summary

You should know the answers to these questions:

- ❑ What is an assertion?
- ❑ How are contracts formalized by pre- and post-conditions?
- ❑ What is a class invariant and how can it be specified?
- ❑ What are assertions useful for?
- ❑ How can exceptions be used to improve program robustness?
- ❑ What situations may cause an exception to be raised?

Can you answer the following questions?

- ✎ *What happens when you pop() an empty java.util.Stack? Is this good or bad?*
- ✎ *What impact do assertions have on performance?*

3. Testing and Debugging

Overview

- ❑ Testing — definitions
- ❑ Testing various Stack implementations
- ❑ Understanding the run-time stack and heap
- ❑ Wrapping — a simple integration strategy
- ❑ Timing benchmarks

Source

- ❑ I. Sommerville, *Software Engineering*,Addison-Wesley, Fifth Edn., 1996.

Testing

1. Unit testing:
   - ☞ test individual (stand-alone) components
2. Module testing:
   - ☞ test a collection of related components (a module)
3. Sub-system testing:
   - ☞ test sub-system interface mismatches
4. System testing:
   - ☞ (i) test interactions between sub-systems, and
     (ii) test that the complete systems fulfils functional and non-functional requirements
5. Acceptance testing (alpha/beta testing):
   - ☞ test system with real rather than simulated data.

Testing is iterative!

Regression testing

Regression testing means testing that everything that used to work *still works* after changes are made to the system!

- ❑ tests must be *deterministic* and *repeatable*
- ❑ should test “all” functionality
  - ☞ every interface
  - ☞ all boundary situations
  - ☞ every feature
  - ☞ every line of code
  - ☞ everything that can conceivably go wrong!

It costs extra work to define tests up front, but they pay off in debugging & maintenance!

NB: Testing can only reveal the *presence* of defects, not their absence!

Stack test case

We define a simple regression test that exercises *all StackInterface methods* and checks the *boundary situations:*

```
assert(stack.isEmpty());

for (int i=1; i<=10; i++) { stack.push(new Integer(i)); }
assert(!stack.isEmpty());
assert(stack.size() == 10);
assert(((Integer) stack.top()).intValue() == 10);

for (int i=10; i>1; i--) { stack.pop(); } // pop 10 .. 2
assert(!stack.isEmpty());
assert(stack.size() == 1);
assert(((Integer) stack.top()).intValue() == 1);

stack.pop();
assert(stack.isEmpty());
```

Testing special cases

We would also like to know that our Stack checks for failed pre-conditions!

```
boolean emptyPopCaught = false;
try {
  // we expect pop() to raise an exception
  stack.pop();
} catch(AssertionException err) {
  // we should get here!
  emptyPopCaught = true;
}
assert(emptyPopCaught); // should be true
```

TestStack

We define a method that will test any given implementation of StackInterface:

```
static public void testStack(StackInterface stack) {
  try {
    System.out.print("Testing "
      + stack.getClass().getName() + " ... ");

    // the tests go here ...

    System.out.println("passed all tests!");
  } catch (Exception err) {                    // NB: any kind!
    err.printStackTrace();
  }
}
```

Running the test yields:

```
Testing LinkStack ... passed all tests!
```

When (not) to use static methods

A *static* method or instance variable belongs to a *class*, not an object.

❑ Static methods can be called without instantiating an object

- necessary for *starting the main program*
- necessary for *constructors* and *factory methods*
- useful for test methods

❑ Static methods are just *procedures!*

☞ avoid them in OO designs!

❑ Static instance variables can be accessed without instantiating an object

- useful for representing data *shared by all instances* of a class

❑ Static variables are *global variables!*

☞ avoid them in OO designs!

ArrayStack

A very different way to implement a Stack is with a (fixed-length) array.

When the array runs out of space, the Stack “grows” by allocating a larger array, and copying elements to the new array

```
public class ArrayStack implements StackInterface {
  Object _store [];
  int _capacity;
  int _size;

  public ArrayStack() {
    _store = null;              // default value
    _capacity = 0;
    _size = 0;
  }
  ...
}
```

ArrayStack methods

```
public boolean isEmpty() { return _size == 0; }
public int size() { return _size; }

public void push(Object item) throws AssertionException {
  if (_size == _capacity) { grow(); }
  _store[++_size] = item;              // NB: subtle error!
}

public Object top() throws AssertionException {
  assert(!this.isEmpty());
  return _store[_size-1];
}

public void pop() throws AssertionException {
  assert(!this.isEmpty());
  _size--;
}
```

NB: we only check pre-conditions in this version!

Testing ArrayStack

```
Testing ArrayStack ...
java.lang.ArrayIndexOutOfBoundsException: 2
  at ArrayStack.push(ArrayStack.java:28)
  at TestStack.testStack(Compiled Code)
  at TestStack.main(TestStack.java:12)
  at com.apple.mrj.JManager.JMStaticMethodDispatcher
      .run(JM-AWTContextImpl.java:796)
  at java.lang.Thread.run(Thread.java:474)
```

Exception.printStackTrace() tells us exactly where the exception occurred ...

The Run-time Stack

The *run-time stack* is a fundamental data structure used to record a *context* of a procedure that will be returned to at a later point in time. This context (AKA “stack frame”) stores the *arguments* to the procedure and its *local variables.*

Practically *all* programming languages use a run-time stack:

```
public static void main(String args[]) {
  System.out.println( "fact(3) = " + fact(3));
}

public static int fact(int n) {
  if (n<=0) {
    return 1;
  } else {
    return n*fact(n-1);
  }
}
```

The run-time stack in action ...

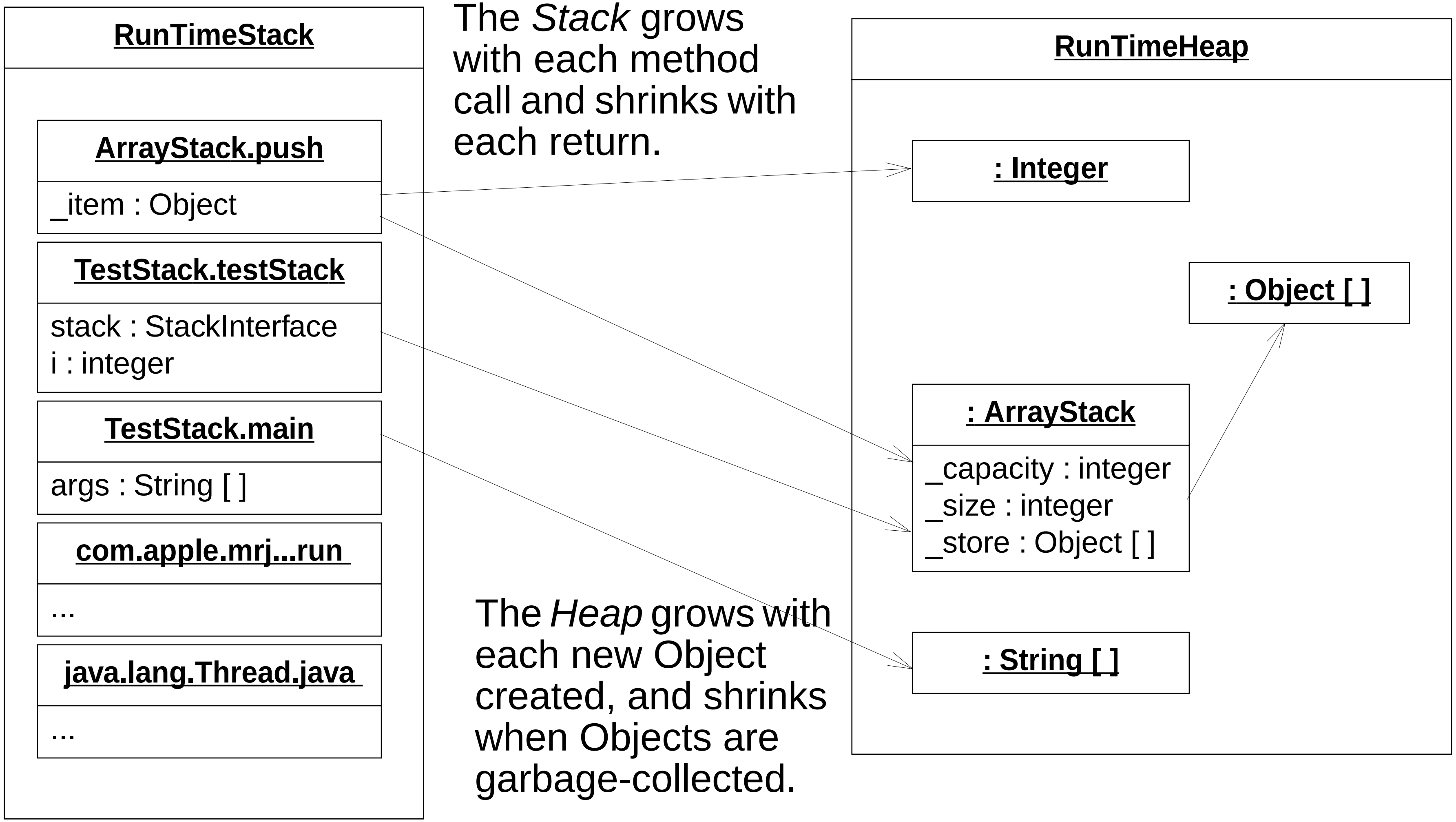
The stack grows with each procedure call ...

`main ...`				
`main;fact(3)=?`	`fact(3) ...`			
`main;fact(3)=?`	`fact(3);fact(2)=?`	`fact(2) ...`		
`main;fact(3)=?`	`fact(3);fact(2)=?`	`fact(2);fact(1)=?`	`fact(1) ...`	
`main;fact(3)=?`	`fact(3);fact(2)=?`	`fact(2);fact(1)=?`	`fact(1);fact(0)=?`	`fact(0) ...`
`main;fact(3)=?`	`fact(3);fact(2)=?`	`fact(2);fact(1)=?`	`fact(1);fact(0)=?`	`fact(0);return 1`
`main;fact(3)=?`	`fact(3);fact(2)=?`	`fact(2);fact(1)=?`	`fact(1);return 1`	
`main;fact(3)=?`	`fact(3);fact(2)=?`	`fact(2);return 2`		
`main;fact(3)=?`	`fact(3);return 6`			
`main;fact(3)=6`				

... and shrinks with each return.

The Stack and the Heap

The *Stack* grows with each method call and shrinks with each return.



The *Heap* grows with each new Object created, and shrinks when Objects are garbage-collected.

Fixing our mistake

We erroneously used the *incremented* _size as an index into the _store, instead of the new size - 1:

```
public void push(Object item) throws AssertionException {
  if (_size == _capacity) { grow(); }
  // NB: top index is the *old* value of _size
  _store[_size++] = item;
  assert(this.top() == item);
  assert(invariant());
}
```

Perhaps it would be clearer to write:

```
_store[this.topIndex()] = item;
```

or even:

```
this.setTop(item)
```

Wrapping Objects

Java also provides a Stack implementation, but it is not compatible with our interface:

```
public class Stack extends Vector {
  public Stack();
  public Object push(Object item);
  public synchronized Object pop();
  public synchronized Object peek();
  public boolean empty();
  public synchronized int search(Object o);
}
```

If we change our programs to work with the Java Stack, we won’t be able to work with our own Stack implementations ...

➤ What do you do with an object whose interface doesn’t fit your expectations?

✔ *You wrap it.*

A Wrapped Stack

Wrapping is a fundamental programming technique for systems integration.

```
import java.util.Stack;
public class SimpleWrappedStack implements StackInterface {
  Stack _stack;
  public SimpleWrappedStack() { _stack = new Stack(); }
  public boolean isEmpty() { return _stack.empty(); }
  public int size() { return _stack.size(); }
  public void push(Object item) throws AssertionException {
    _stack.push(item);
  }
  public Object top() throws AssertionException {
    return _stack.peek();
  }
  public void pop() throws AssertionException {
    _stack.pop();
  }
}
```

✎ *What are possible disadvantages of wrapping?*

A contract mismatch

But running `testStack(new SimpleWrappedStack())` yields:

```
Testing SimpleWrappedStack ... java.util.EmptyStackException
  at java.util.Stack.peek(Stack.java:78)
  at java.util.Stack.pop(Stack.java:60)
  at SimpleWrappedStack.pop(SimpleWrappedStack.java:29)
  at TestStack.testStack(Compiled Code)
  at TestStack.main(TestStack.java:13)
  at com.apple.mrj.JManager.JMStaticMethodDispatcher.
      run(JMAWTContextImpl.java:796)
  at java.lang.Thread.run(Thread.java:474)
```

Fixing the problem ...

Our tester *expects* an empty Stack to throw an exception when it is popped, but java.util.Stack doesn’t do this — so our wrapper should check its preconditions!

```
public class WrappedStack extends SimpleWrappedStack {
  public Object top() throws AssertionException {
    assert(!this.isEmpty());
    return super.top();
  }
  public void pop() throws AssertionException {
    assert(!this.isEmpty());
    super.pop();
  }
  private void assert(boolean assertion)
    throws AssertionException { ... }
}
```

Timing benchmarks

Which of the Stack implementations performs better?

```
timer.reset();
for (int i=0; i<iterations; i++) {
  stack.push(item);
}
elapsed = timer.timeElapsed();
System.out.println(elapsed + " milliseconds for "
    + iterations + " pushes");
...
```

➤ Complexity aside, how can you tell which implementation strategy will perform best?

✔ *Run a benchmark.*

Timer

We can abstract from the details of how to obtain the timings:

```
import java.util.Date;
public class Timer {
   long _startTime;
   public Timer() { this.reset(); }
   public void reset() {
      _startTime = this.timeNow();
   }
   public long timeElapsed() {
      return this.timeNow() - _startTime;
   }
   protected long timeNow() {
      return new Date().getTime();
   }
}
```

✎ *What would the benchmark routine look like without using the Timer abstraction?*

Sample benchmarks

Times are in milliseconds ...

Java VM	*Stack Implementation*	*100K pushes*	*100K pops*
Apple MRJ	LinkStack	2809	100
	ArrayStack	474	56
	WrappedStack	725	293
Metrowerks	LinkStack	5151	1236
	ArrayStack	1519	681
	WrappedStack	8748	8249
Metrowerks JIT	LinkStack	3026	189
	ArrayStack	877	94
	WrappedStack	5927	5318

✎ *Can you explain these results? Are they what you expected?*

✎ *What happens if you run these tests several times?*

Summary

You should know the answers to these questions:

- ❑ What is a regression test? Why is it important?
- ❑ What strategies should you apply to design a test?
- ❑ What are the run-time stack and heap?
- ❑ How can you adapt client/supplier interfaces that don’t match?
- ❑ When are benchmarks useful?

Can you answer the following questions?

- ✎ *How would you implement ArrayStack.grow()?*
- ✎ *What are the advantages and disadvantages of wrapping?*
- ✎ *What is a suitable class invariant for WrappedStack?*
- ✎ *How can we learn where each Stack implementation is spending its time?*
- ✎ *How much can the same benchmarks differ if you run them several times?*

4. Iterative Development

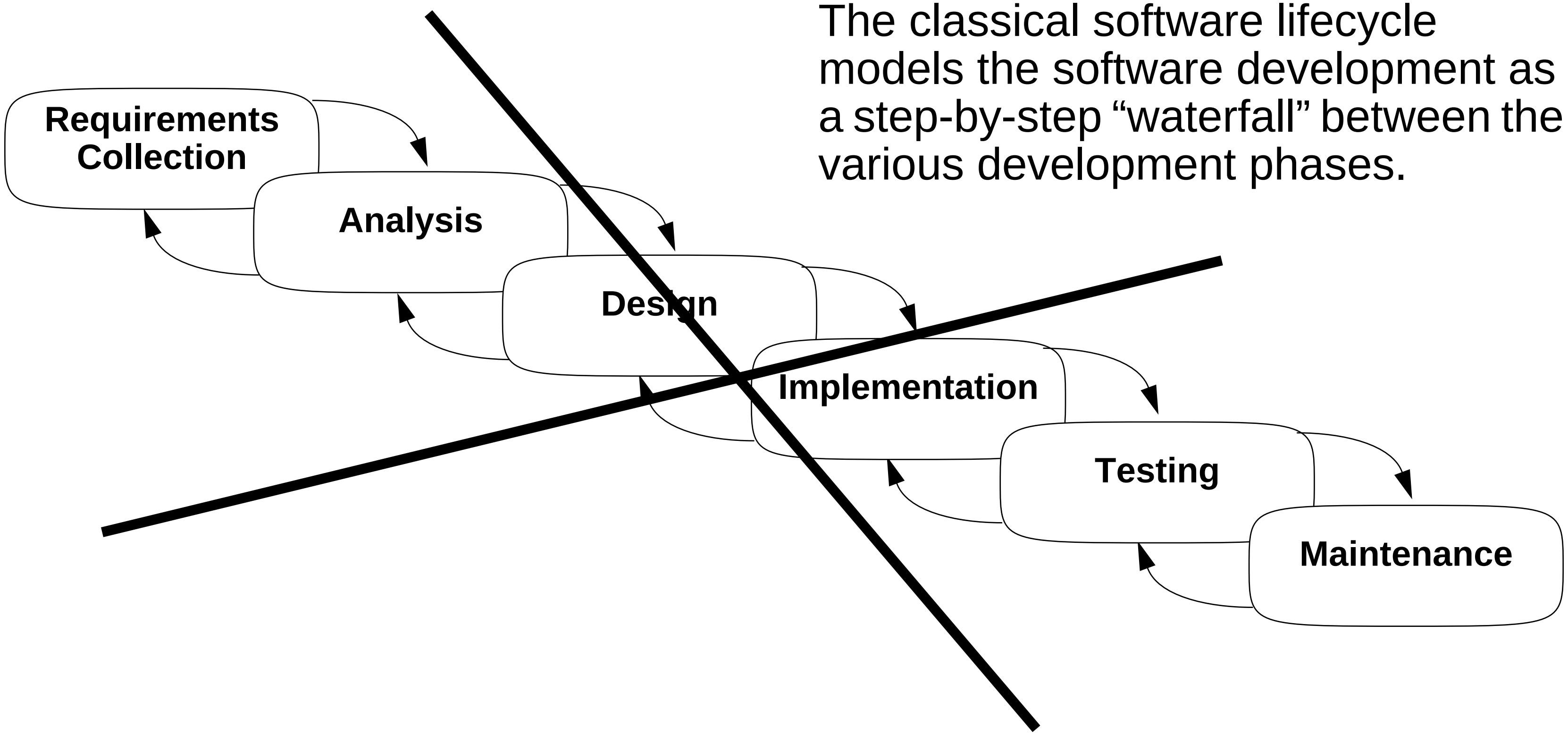
Overview

- ❑ Iterative development
- ❑ Responsibility-Driven Design
  - ☞ How to find the objects ...
  - ☞ TicTacToe example ...

Sources

- ❑ R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.
- ❑ Kent Beck, *Embrace Change: Extreme Programming Explained,* draft manuscript, 1999.

The Classical Software Lifecycle



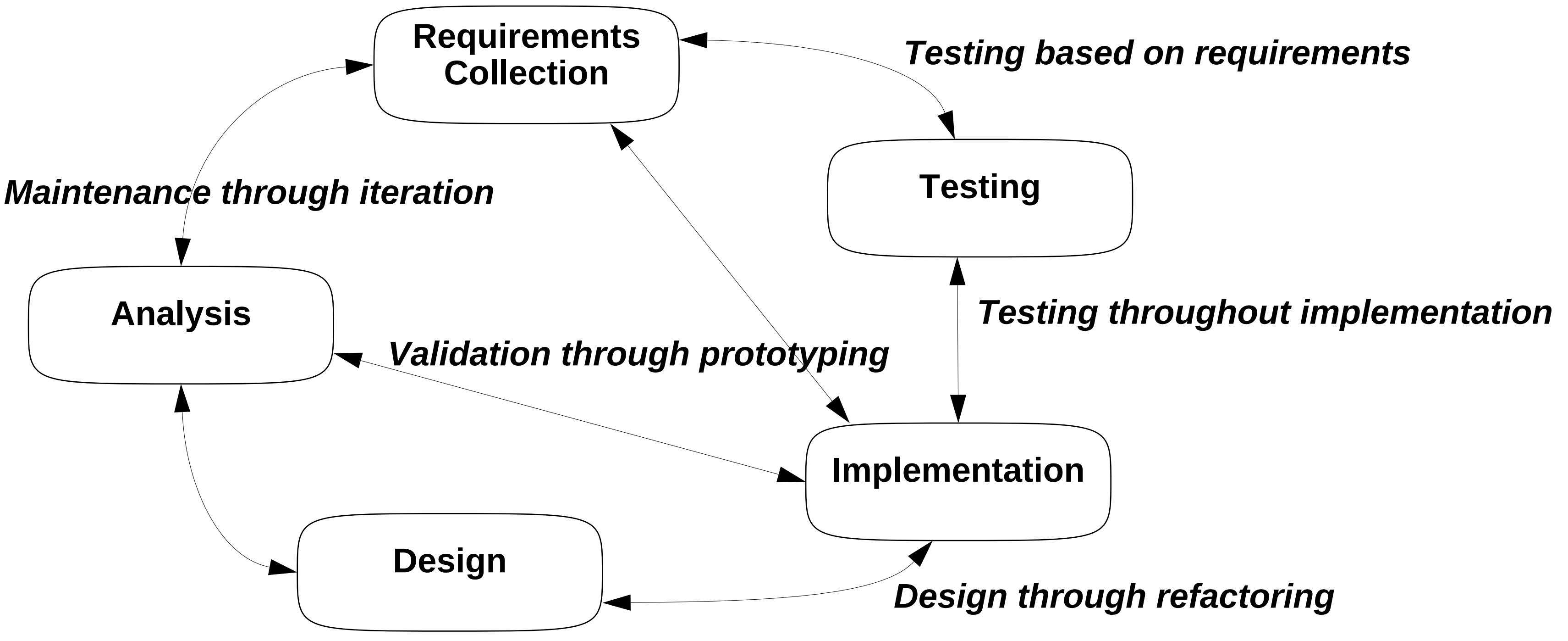
The classical software lifecycle models the software development as a step-by-step “waterfall” between the various development phases.

The waterfall model is unrealistic for many reasons, especially:

- ❑ requirements must be “frozen” too early in the life-cycle
- ❑ requirements are validated too late

Iterative Development

In practice, development is always iterative, and *all* software phases progress in parallel.


Requirements Collection
Testing based on requirements
Testing
Maintenance through iteration
Analysis
Testing throughout implementation
Validation through prototyping
Implementation
Design
Design through refactoring

✎ *If the waterfall model is pure fiction, why is it still the standard software process?*

What is Responsibility-Driven Design?

Responsibility-Driven Design is

- ❑ a method for deriving a software design in terms of *collaborating objects*
- ❑ by asking what *responsibilities* must be fulfilled to meet the requirements,
- ❑ and assigning them to the *appropriate objects* (i.e., that can carry them out).

Pelrine’s Laws

➤ How do you decide what responsibilities to assign to an object?

✔ *“Don't do anything you can push off to someone else.”*

➤ When should you let an object export its state?

✔ *“Don't let anyone else play with you.”*

RDD leads to fundamentally different designs than those obtained by functional decomposition or data-driven design.

☞ class responsibilities tend to be more stable over time than functionality or representation

Example: Tic Tac Toe

Requirements: [Random House Dictionary of the English Language]

> *“A simple game in which one player marks down only crosses and another only ciphers [zeroes], each alternating in filling in marks in any of the nine compartments of a figure formed by two vertical lines crossed by two horizontal lines, the winner being the first to fill in three of his marks in any row or diagonal.”*

We should design a program that implements the rules of Tic Tac Toe.

Limiting Scope

Questions:

- ❑ Should we support other games?
- ❑ Should there be a graphical UI?
- ❑ Should games run on a network? Through a browser?
- ❑ Can games be saved and restored?

A monolithic paper design is bound to be wrong!

An iterative development strategy:

- ❑ reduce scope to the *minimal requirements* that are interesting
- ❑ *grow the system* by adding features and test cases
- ❑ let the design emerge by *refactoring* roles and responsibilities

➤ How much functionality should you deliver in the first version of a system?

✔ *Select the minimal requirements that provide value to the client.*

Tic Tac Toe Objects

We (may) first try to identify likely objects occurring in the requirements:

Objects	*Responsibilities*
Game	Maintain game rules
Player	Make moves Mediate user interaction
Compartment	Record marks
Figure (State)	Maintain game state

Non-Objects	*Justification*
Crosses, ciphers	Same as Marks
Marks	Value of Compartment
Vertical lines	Display of State
Horizontal lines	ditto
Winner	State of Player
Row	View of State
Diagonal	ditto

Entities with clear responsibilities are more likely to end up as objects in our design.

➤ How can you tell when you have the "right" set of objects?

✔ *Each object has a clear and natural set of responsibilities.*

Missing Objects

At this point we can ask if there are unassigned responsibilities:

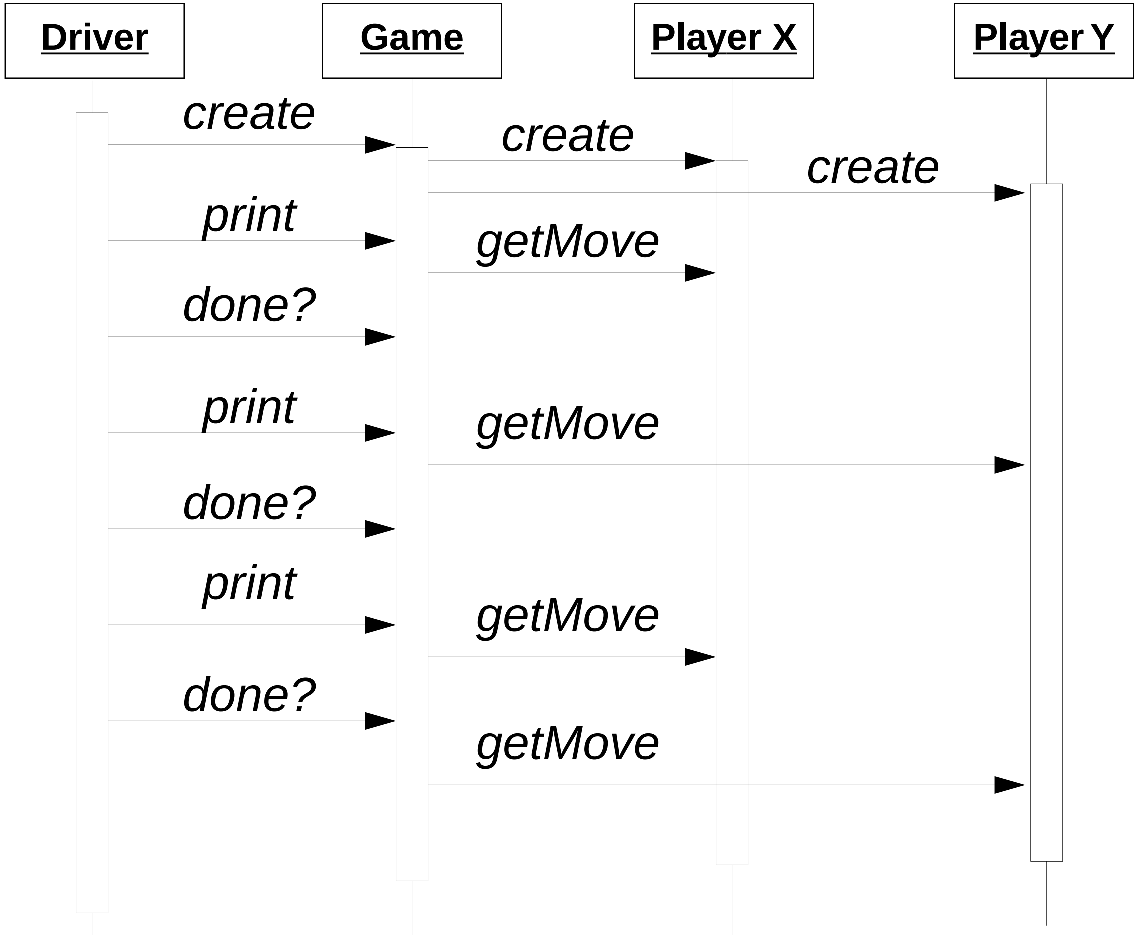
- ❑ Who starts the Game?
- ❑ Who is responsible for displaying the Game state?
- ❑ How do Players know when the Game is over?

Let us introduce a *Driver* that supervises the Game.

➤ How can you tell when there are objects missing in your design?
✔ *When there are responsibilities that cannot be assigned to some object.*

Scenarios

A *scenario* describes a typical sequence of interactions between objects:



✎ *Can you imagine other, equally valid scenarios for the same problem?*

Version 1.0 (skeleton)

Our first version does very little!

```
class GameDriver {
  static public void main(String args[]) {
    TicTacToe game = new TicTacToe();
    do {
      System.out.print(game);
    } while(game.notOver());
  }
public class TicTacToe {
  public boolean notOver() { return false; }
  public String toString() { return("TicTacToe\n"); }
}
```

➤ How do you iteratively “grow” a program?

✔ *Always have a running version of your program.*

Version 1.1 (simple tests)

The state of the game is represented as 3x3 array of chars marked ‘ ’, ‘X’, or ‘O’. We index the state using chess notation, i.e., column is 'a' through 'c' and row is '1' through '3'.

```
public class TicTacToe {
  private char[][] _gameState;

  public TicTacToe() {
    _gameState = new char[3][3];
    for (char col='a'; col <='c'; col++)
      for (char row='1'; row<='3'; row++)
        this.set(col,row,' ');
  }
  private void set(char col, char row, char mark) {
    assert(inRange(col, row)); // NB: precondition
    _gameState[col-'a'][row-'1'] = mark;
  }
  private char get(char col, char row) { ... }
... }
```

Testing the new methods

For now, our tests can just exercise the new set() and get() methods:

```
  public void test() {
    System.err.println("Started TicTacToe tests");
    assert(this.get('a','1') == ' ');
    assert(this.get('c','3') == ' ');
    this.set('c','3','X');
    assert(this.get('c','3') == 'X');
    this.set('c','3',' ');
    assert(this.get('c','3') == ' ');
    assert(!this.inRange('d','4'));
    System.err.println("Passed TicTacToe tests");
  }
}
```

Testing the application

If each class provides its own test() method, we can bundle our unit tests in a single driver class:

```
class TestDriver {
  static public void main(String args[]) {
    TicTacToe game = new TicTacToe();
    game.test();
  }
}
```

Printing the State

By re-implementing `TicTacToe.toString()`, we can view the state of the game:

```
3    |   |
  ---+---+---
2    |   |
  ---+---+---
1    |   |
   a   b   c
```

➤ How do you make an object printable?

✔ *Override Object.toString()*

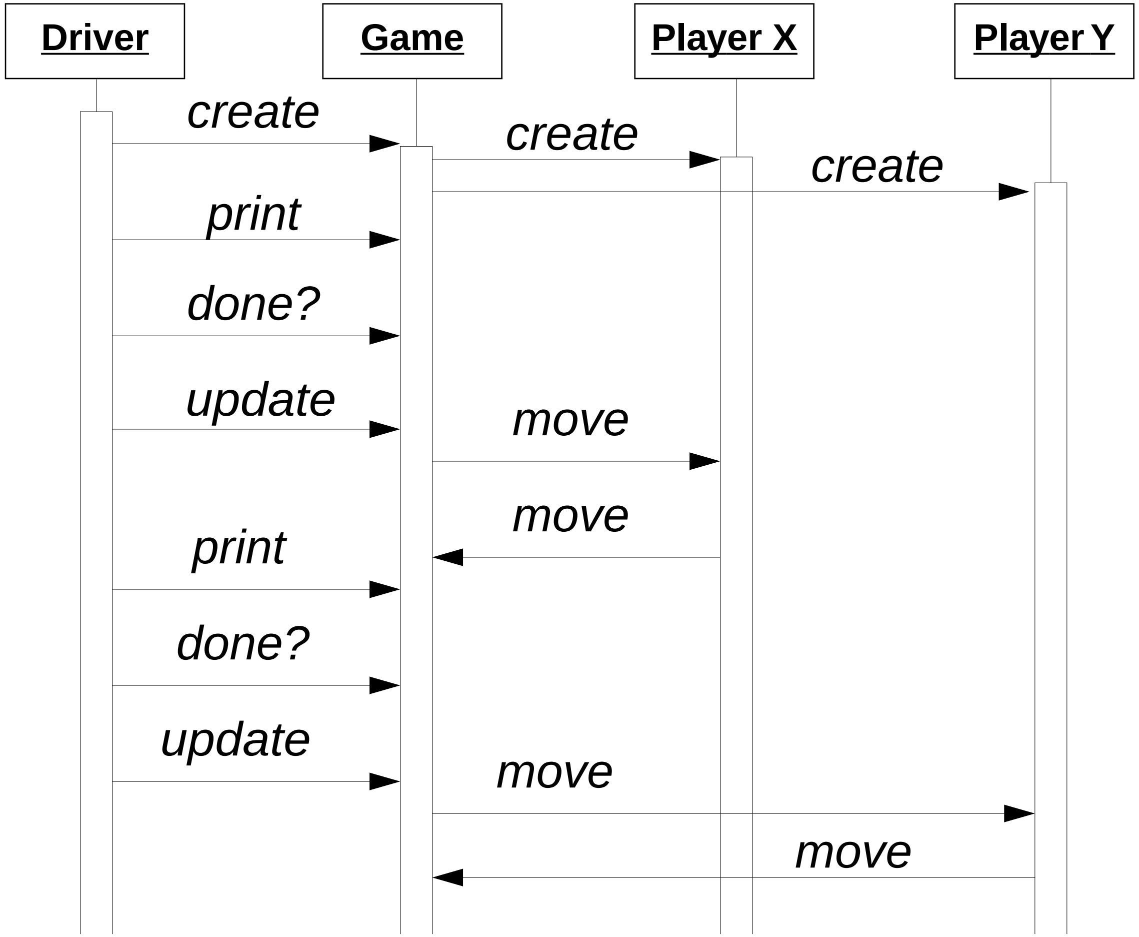
TicTacToe.toString()

Use a StringBuffer (not a String) to build up the representation:

```
public String toString() {
  StringBuffer rep = new StringBuffer();
  for (char row='3'; row>='1'; row--) {
    rep.append(row);
    rep.append("  ");
    for (char col='a'; col <='c'; col++) { ... }
    ...
  }
  rep.append("   a   b   c\n");
  return(rep.toString());
}
```

Refining the interactions

We see now that updating the Game and printing it should be separate operations:



The Game can ask the Player to make a move, and the Player will attempt to do so ...

Tic Tac Toe Contracts

Consider all the assertions that should hold at various points in a game ...

Explicit invariants:

- turn (current player) is either X or O
- X and O swap turns (turn never equals previous turn)
- game state is 3×3 array marked X, O or blank
- winner is X or O iff winner has three in a row

Implicit invariants:

- initially winner is nobody; initially it is the turn of X
- game is over when all squares are occupied, or there is a winner
- a player cannot mark a square that is already marked

Contracts:

- the current player may make a move, if the invariants are respected

Version 1.2 (functional)

We must introduce state variables to implement the contracts

```
public class TicTacToe {
  // ...
  private Player _winner = new Player(); // represents nobody
  private Player[] _player;
  private int _turn = X;                 // initial turn
  private int _squaresLeft = 9;
  static final int X = 0;
  static final int O = 1;

  public TicTacToe(Player playerX, Player playerO)
    throws AssertionException
  { // ...
    _player = new Player[2];
    _player[X] = playerX;
    _player[O] = playerO;
  }
```

Invariants

Since invariants must hold at the end of each method, it can be useful to define a separate method.

These conditions seem obvious, which is exactly why they should be checked ...

```
private boolean invariant() {
  return (_turn == X || _turn == O)
    && (this.notOver()
       || this.winner() == _player[X]
       || this.winner() == _player[O]
       || this.winner().isNobody())
    && (_squaresLeft < 9      // else, initially:
       || _turn == X && this.winner().isNobody());
}
```

Assertions and tests often tell us what methods should be implemented, and whether they should be public or private.

Delegating Responsibilities

When Driver updates the Game, the Game just asks the Player to make a move:

```
public void update() throws IOException {
    _player[_turn].move(this);
}
```

The Game also has a move() method, called when the Player makes its move:

```
public void move(char col, char row, char mark)
    throws AssertionException
{
    assert(this.notOver());
    assert(inRange(col, row));
    assert(this.get(col, row) == ' ');
    System.out.println(mark + " at " + col + row);
    this.set(col, row, mark);
    this._squaresLeft--;
    this.swapTurn();
    this.checkWinner();
    assert(this.invariant());
}
```

Small Methods

Well-named variables and methods typically eliminate the need for explanatory comments!

Introduce methods that make the *intent* of your code clear.

```
public boolean notOver() {
  return this.winner().isNobody()
    && this.squaresLeft() > 0;
}
private void swapTurn() { _turn = (_turn == X) ? O : X; }

public Player winner() { return _winner; }
public int squaresLeft() { return this._squaresLeft; }
```

➤ When should instance variables be public?

✔ *Almost never! Declare public accessor methods instead.*

Code Smells — TicTacToe.checkWinner()

Check for a winning row, column or diagonal:

```
private void checkWinner()
   throws AssertionException
{
   char player;
   for (char row='3'; row>='1'; row--) {
      player = this.get('a',row);
      if (player == this.get('b',row)
         && player == this.get('c',row)) {
         this.setWinner(player);
         return;
      }
   }
   for (char col='a'; col <='c'; col++) {
      player = this.get(col,'1');
      if (player == this.get(col,'2')
         && player == this.get(col,'3')) {
         this.setWinner(player);
         return;
      }
   }
   player = this.get('b','2');
   if (player == this.get('a','1')
      && player == this.get('c','3')) {
         this.setWinner(player);
         return;
   }
   if (player == this.get('a','3')
      && player == this.get('c','1')) {
         this.setWinner(player);
         return;
   }
}
```

✎ *This code smells! How can we clean it up?*

GameDriver

In order to run test games, we must *separate Player instantiation from Game playing:*

```
public class GameDriver {
  public static void main(String args[]) {
    try {
      Player X = new Player('X');
      Player O = new Player('O');
      TicTacToe game = new TicTacToe(X, O);
      playGame(game);
    } catch (AssertionException err) {
      ...
    }
  }
```

The Player

Multiple constructors are needed to distinguish real and virtual Players:

```
public class Player {
  private final char _mark;
  private final BufferedReader _in;

  public Player(char mark, BufferedReader in) { // internal
    _mark = mark;
    _in = in;
  }
  public Player(char mark) { // the normal constructor
    this(mark,
      new BufferedReader(new InputStreamReader(System.in)));
  }
  public Player(char mark, String moves) { // for testing
    this(mark, new BufferedReader(new StringReader(moves)));
  }
  public Player() { this(' '); } // for Player “nobody”
```

Defining test cases

```
public class TestDriver {
  private static String testX1 = "a1\nb2\nc3\n";
  private static String testO1 = "b1\nc1\n";
  // + other test cases ...
  public static void main(String args[]) {
    testGame(testX1, testO1, "X", 4); // ...
  }
  public static void testGame(String Xmoves, String Omoves,
      String winner, int squaresLeft) {
    try {
      Player X = new Player('X', Xmoves);
      Player O = new Player('O', Omoves);
      TicTacToe game = new TicTacToe(X, O);
      GameDriver.playGame(game);
      assert(game.winner().name().equals(winner));
      assert(game.squaresLeft() == squaresLeft);
    } catch (AssertionException err) { ... }
  }
```

Running the test cases

```
Started testGame test
3    |   |
  ---+---+---
2    |   |
  ---+---+---
1    |   |
   a   b   c
Player X moves: X at a1
3    |   |
  ---+---+---
2    |   |
  ---+---+---
1  X |   |
   a   b   c
Player O moves: O at b1
3    |   |
  ---+---+---
2    |   |
  ---+---+---
1  X | O |
   a   b   c
```

```
Player X moves: X at b2
3    |   |
  ---+---+---
2    | X |
  ---+---+---
1  X | O |
   a   b   c
Player O moves: O at c1
3    |   |
  ---+---+---
2    | X |
  ---+---+---
1  X | O | O
   a   b   c
Player X moves: X at c3
3    |   | X
  ---+---+---
2    | X |
  ---+---+---
1  X | O | O
   a   b   c
game over!
Passed testGame test
```

Summary

You should know the answers to these questions:

- ❑ What is Iterative Development, and how does it differ from the Waterfall model?
- ❑ How can identifying responsibilities help you to design objects?
- ❑ Where did the Driver come from, if it wasn’t in our requirements?
- ❑ Why is Winner not a likely class in our TicTacToe design?
- ❑ Why should we evaluate assertions if they are all supposed to be true anyway?
- ❑ What is the point of having methods that are only one or two lines long?

Can you answer the following questions?

- ✎ *Why should you expect requirements to change?*
- ✎ *In our design, why is it the Game and not the Driver that prompts a Player to move?*
- ✎ *When and where should we evaluate the TicTacToe invariant?*
- ✎ *What other tests should we put in our TestDriver?*
- ✎ *How does the Java compiler know which version of an overloaded method or constructor should be called?*

5. Inheritance and Refactoring

Overview

- ❑ Uses of inheritance
  - ☞ conceptual hierarchy, polymorphism and code reuse
- ❑ TicTacToe and Gomoku
  - ☞ which inherits from which?!
  - ☞ interfaces and abstract classes
- ❑ Refactoring
  - ☞ iterative strategies for improving design
- ❑ Top-down decomposition
  - ☞ decompose algorithms into high-level steps to reduce complexity
  - ☞ use recursion when it can simplify your design

Source

- ❑ R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

What is Inheritance?

Inheritance in object-oriented programming languages is a *mechanism* to:

- ❑ *derive new subclasses* from existing classes
- ❑ where subclasses *inherit all the features* from their parent(s)
- ❑ and may selectively *override* the implementation of some features.

Various OOPLs may additionally provide:

- ❑ *self* — a way to dynamically access methods of the current instance
- ❑ *super* — a way to statically access overridden, inherited methods
- ❑ *multiple inheritance* — a way to inherit features of multiple classes
- ❑ *abstract classes* — partially defined classes (to inherit from only)
- ❑ *mixins* — a way to build classes from partial sets of features
- ❑ *interfaces* — specifications of method argument and return types only
- ❑ *subtyping* — guarantees that subclass instances can be substituted for their parents
- ❑ ...

The Board Game

Tic Tac Toe is a pretty dull game, but there are many other interesting games that can be played by two players with a board and two colours of markers.

Example: Go-moku [Random House Dictionary of the English Language]

> *“A Japanese game played on a go board with players alternating and attempting to be first to place five counters in a row.”*

☞ We would like to implement a program that can be used to play several different kinds of games using the same game-playing abstractions (starting with TicTacToe and Go-moku).

Uses of Inheritance

Inheritance in object-oriented programming languages can be used for (at least) three different, but closely related purposes:

Conceptual hierarchy:

- ❑ Go-moku *is-a* kind of Board Game; Tic Tac Toe *is-a* kind of Board Game

Polymorphism:

- ❑ Instances of `Gomoku` and `TicTacToe` can be uniformly manipulated as instances of `BoardGame` by a client program

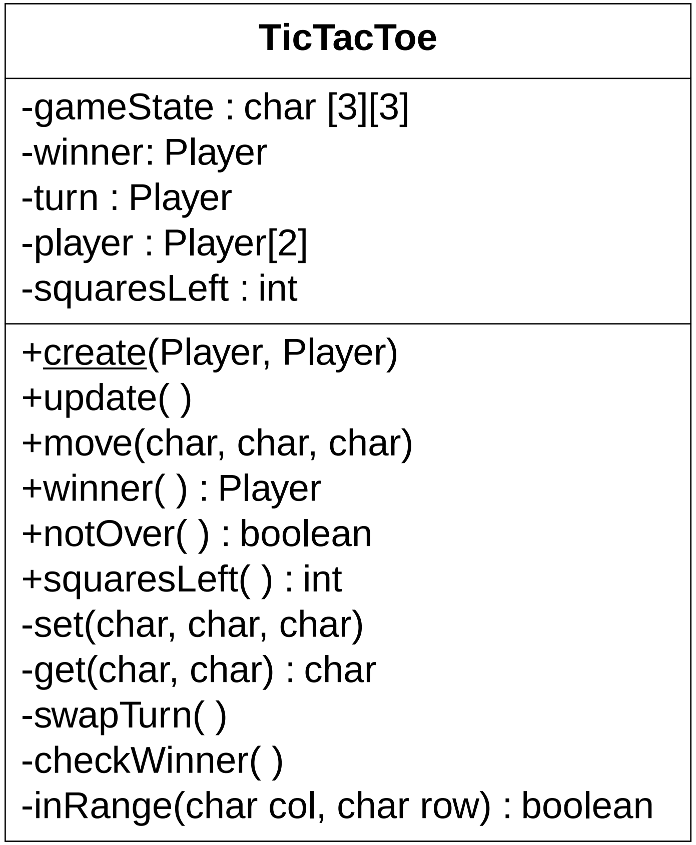
Software reuse:

- ❑ `Gomoku` and `TicTacToe` reuse the `BoardGame` interface
- ❑ `Gomoku` and `TicTacToe` reuse and extend the `BoardGame` representation and the implementations of its operations

Class Diagrams

At this stage the key classes look like this:

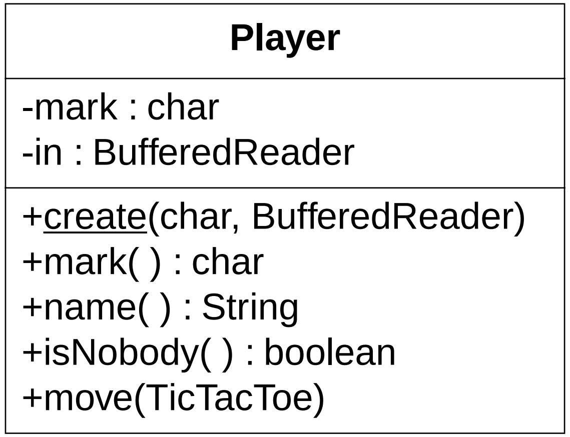
Key	
-	private feature
#	protected feature
+	public feature
create( )	static feature
checkWinner( )	abstract feature



TicTacToe

-gameState : char [3][3]
-winner: Player
-turn : Player
-player : Player[2]
-squaresLeft : int

+create(Player, Player)
+update( )
+move(char, char, char)
+winner( ) : Player
+notOver( ) : boolean
+squaresLeft( ) : int
-set(char, char, char)
-get(char, char) : char
-swapTurn( )
-checkWinner( )
-inRange(char col, char row) : boolean



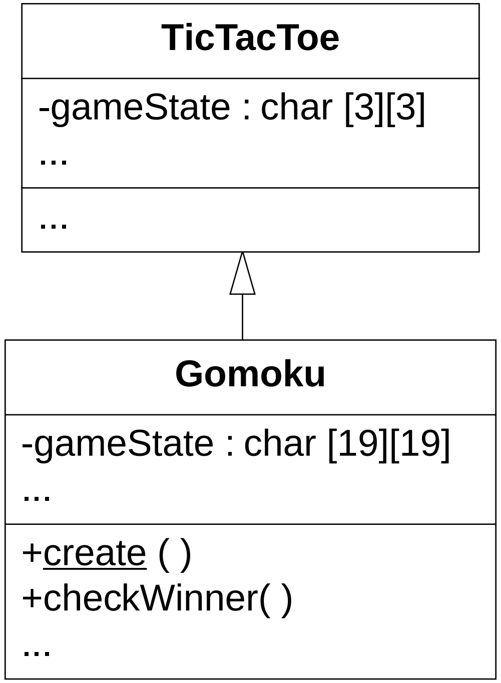
Player

-mark : char
-in : BufferedReader

+create(char, BufferedReader)
+mark( ) : char
+name( ) : String
+isNobody( ) : boolean
+move(TicTacToe)

A bad idea ...

Why not simply use inheritance for incremental modification?



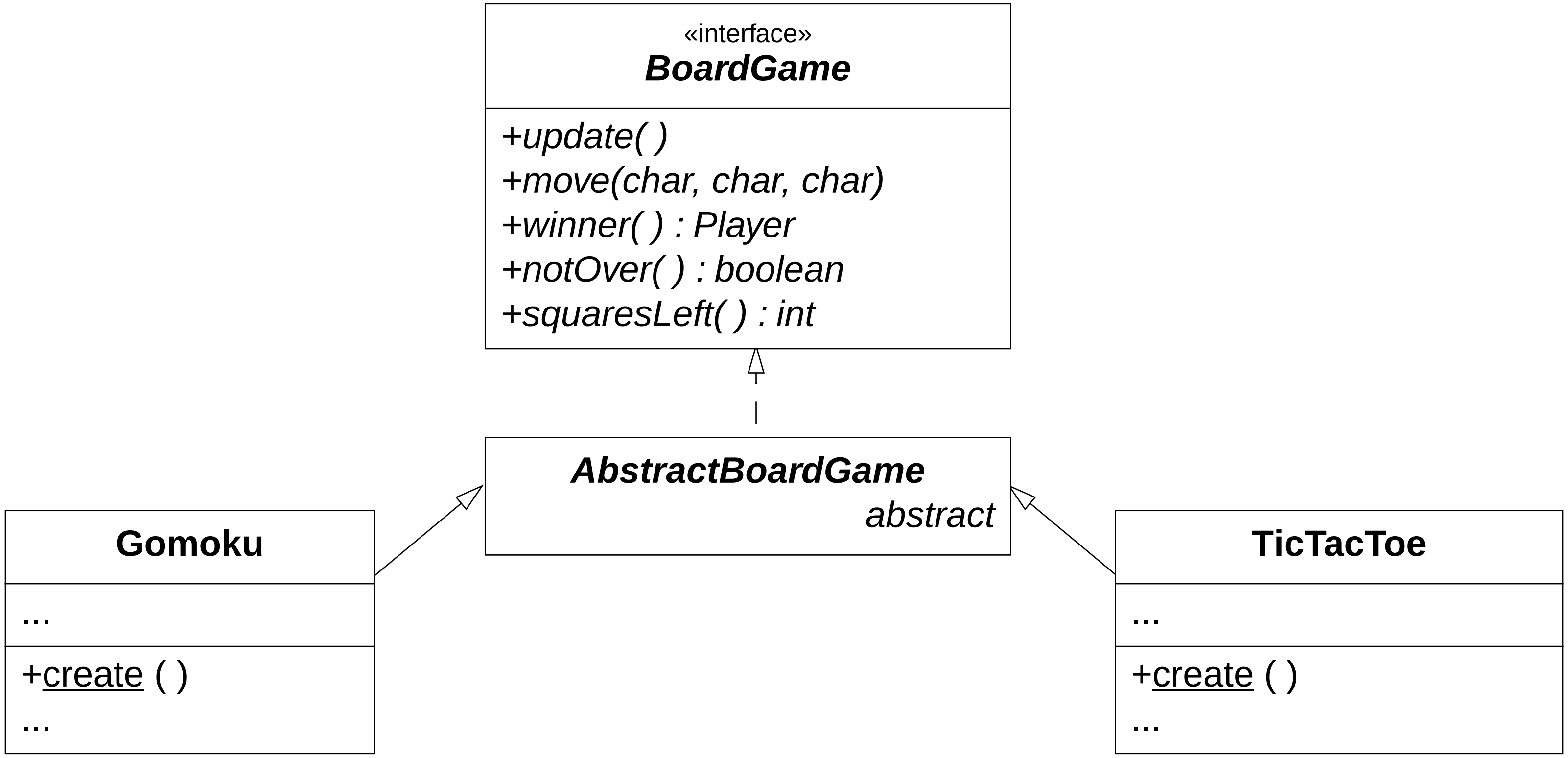
Exploiting inheritance for code reuse *without refactoring* tends to lead to:

- ❑ duplicated code *(similar, but not reusable methods)*
- ❑ conceptually unclear design *(arbitrary relationships between classes)*

Gomoku is not a kind of TicTacToe

Class Hierarchy

Both Go-moku and Tic Tac Toe are kinds of Board games (IS-A). We would like to define a common interface, and factor the common functionality into a shared parent class.



Behaviour that is not shared will be implemented by the subclasses.

Iterative development strategy

We need to find out which TicTacToe functionality will:

- ❑ *already work* for both TicTacToe and Gomoku
- ❑ need to be *adapted* for Gomoku
- ❑ can be *generalized* to work for both

Example: `set()` and `get()` will not work for a 19×19 board!

Rather than attempting a “big bang” redesign, we will *iteratively redesign* our game:

- ❑ introduce a BoardGame interface that TicTacToe implements
- ❑ move all TicTacToe implementation to an AbstractBoardGame parent
- ❑ fix, refactor or make abstract the non-generic features
- ❑ introduce Gomoku as a concrete subclass of AbstractBoardGame

After each iteration we run our regression tests to make sure nothing is broken!

➤ When should you run your (regression) tests?

✔ *After every change to the system.*

Version 1.3 (add interface)

The BoardGame interface specifies the methods that both TicTacToe and Gomoku should implement:

```
public interface BoardGame {
  public void update() throws IOException;
  public void move(char col, char row, char mark)
    throws AssertionException;
  public Player currentPlayer(); // NB: new method needed
  public Player winner();
  public boolean notOver();
  public int squaresLeft();
  public void test();
}
```

Initially we focus only on abstracting from the current TicTacToe implementation

Speaking to an Interface

Clients of TicTacToe and Gomoku should only speak to the BoardGame interface:

```
public class GameDriver {
  public static void main(String args[]) {
    try {
      Player X = new Player('X');
      Player O = new Player('O');
      TicTacToe game = new TicTacToe(X, O);
      playGame(game);
      ...
  }
  public static void playGame(BoardGame game) {
    ...
  }
```

In general, you should *speak to an interface, not an implementation*.

Quiet Testing

Our current TestDriver uses the GameDriver's playGame(), which prints out the state of the game *after each move,* making it hard to tell when a test has failed.

Tests should be silent unless an error has occurred!

```
public static void playGame(BoardGame game) {
  playGame(game, true);
}
public static void playGame(BoardGame game, boolean verbose) {
    ...
      if (verbose) {
        System.out.println();
        System.out.println(game);
        System.out.print("Player "
          + game.currentPlayer().mark() + " moves: ");
  ...
}
```

NB: we must shift all responsibility for printing to playGame().

TicTacToe adaptations

In order to pass responsibility for printing to the GameDriver, TicTacToe must provide a method to export the current Player:

```
public class TicTacToe implements BoardGame {

  ...

  public Player currentPlayer() {
    return _player[_turn];
  }
```

Now we run our regression tests and (after fixing any bugs) continue.

Version 1.4 (add abstract class)

AbstractBoardGame will hold provide common methods for TicTacToe and Gomoku.

```
public abstract class AbstractBoardGame implements BoardGame {
  protected char[][] _gameState;
  protected Player _winner = new Player();// nobody
  protected Player[] _player;
  protected int _turn = X;                      // initial turn
  protected int _squaresLeft = 9;
    ...
  protected void set(char col, char row, char mark)
    ...
  protected char get(char col, char row)
    ...
```

➤ When should a class be declared abstract?

✔ *Declare a class abstract if it is intended to be subclassed, but not instantiated.*

An abstract class may declare abstract methods to be implemented by subclasses ...

Refactoring

Refactoring is a process of moving methods and instance variables from one class to another to improve the design, specifically to:

- ❑ reassign responsibilities
- ❑ eliminate duplicated code
- ❑ reduce coupling
  - ☞ interaction between classes
- ❑ increase cohesion
  - ☞ interaction within classes

We have adopted *one possible refactoring strategy,* first moving everything *except the constructor* from TicTacToe to AbstractBoardGame, and changing all private features to protected:

```
public class TicTacToe extends AbstractBoardGame {
  public TicTacToe(Player playerX, Player playerO)
    ...
```

We could equally have started with an empty AbstractBoardGame ...

Version 1.5 (refactor for reusability)

Now we must check which parts of AbstractBoardGame are generic, which must be repaired, and which must be deferred to its subclasses:

- ❑ the number of rows and columns and the winning score may vary
  - ☞ introduce instance variables and an init() method
  - ☞ rewrite toString(), invariant(), inRange() and test()
- ❑ set() and get() are inappropriate for a 19×19 board
  - ☞ index directly by integers
  - ☞ fix move() to take String argument (e.g., “f17”)
  - ☞ add methods to parse String into integer coordinates
- ❑ getWinner() must be completely rewritten ...

AbstractBoardGame 1.5

We introduce an init() method for arbitrary sized boards:

```
public abstract class AbstractBoardGame implements BoardGame {
  protected void init(int rows, int cols, int score,
    Player playerX, Player playerO)
  { ... }
```

And call it from the constructors of our subclasses:

```
public class TicTacToe extends AbstractBoardGame {
  public TicTacToe(Player playerX, Player playerO)
  {
    // 3x3 board with winning score = 3
    this.init(3,3,3,playerX, playerO);
  }
}
```

✎ *Why not just introduce a constructor for AbstractBoardGame?*

BoardGame 1.5

Most of the changes in AbstractBoardGame are to protected methods.

The only public (interface) method to change is move():

```
public interface BoardGame {
  ...
  public void move(String coord, char mark)
    throws AssertionException;
  ...
}
```

Player 1.5

The Player class is now radically simplified:

```
public class Player {
  ...
  public void move(BoardGame game) throws IOException {
    String line = _in.readLine();
    if (line == null)
      throw new IOException("end of input");
    try {
      game.move(line, this.mark());
    } catch (AssertionException err) {
      System.err.println("Invalid move ignored ("
        + line + ")");
    }
  }
}
```

✎ *How can we make the Player responsible for checking if the move is valid?*

Version 1.6 (Gomoku)

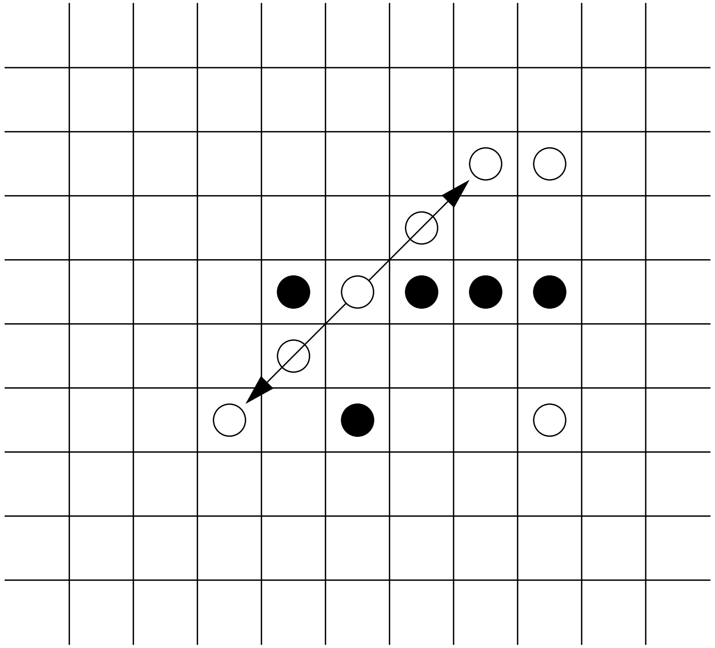
The final steps are:

- ❑ rewrite checkWinner()
- ❑ introduce Gomoku
  - ☞ modify TestDriver to run tests for both TicTacToe and Gomoku
  - ☞ print game state whenever a test fails
- ❑ modify GameDriver to query user for either TicTacToe or Gomoku

Keeping Score

The Go board is too large to search it exhaustively for a winning Go-moku score.

Instead, we know a winning sequence must include *the last square marked,* so we should search in all directions *starting from that square* to see if we find 5 in a row:



We must do the same thing in all four directions.

✎ *Whose responsibility is it to search?*

A new responsibility ...

Maintaining the state of the board and searching for a winning run seem to be unrelated responsibilities. Let’s introduce a separate object whose (called a Runner), whose job it is to run across the board in a given direction and count a Player’s pieces:

```
protected void checkWinner(int col, int row) // NB: new args
  throws AssertionException
{
  char player = this.get(col,row);
  Runner runner = new Runner(this, col, row);
  // check vertically
  if (runner.run(0,1) >= this._winningScore)
    { this.setWinner(player); return; }
  // check horizontally
  if (runner.run(1,0) >= this._winningScore)
    { this.setWinner(player); return; }
  ...
}
```

The Runner

The Runner must know its game, its home (start) position, and its current position:

```
public class Runner {
  BoardGame _game;
  // Home col and row:
  int _homeCol;
  int _homeRow;
  // Current col & row:
  int _col=0;
  int _row=0;

  public Runner(BoardGame game, int col, int row)
  {
    _game = game;
    _homeCol = col;
    _homeRow = row;
  }
...
```

Top-down decomposition

A good way of implementing an algorithm is to describe it *in the most abstract terms possible,* introducing new methods for each abstract step, until you are done:

A runner starts at some home position, runs forward and runs backwards in some direction (delta col and row), adding up a run of tokens of the same kind:

```
public int run(int dcol, int drow) throws AssertionException
{
  int score = 1;
  this.goHome();
  score += this.forwardRun(dcol, drow);
  this.goHome();
  dcol = -dcol;        // reverse direction
  drow = -drow;
  score += this.forwardRun(dcol, drow);
  return score;
}
private void goHome() { _col= _homeCol; _row = _homeRow; }
```

Recursion

Many algorithms are more naturally expressed with recursion than iteration.

Recursively move forward as long as we are in a run. Return the length of the run:

```
private int forwardRun(int dcol, int drow)
  throws AssertionException
{
  this.move(dcol, drow);
  if (this.samePlayer())
    return 1 + this.forwardRun(dcol, drow);
  else
    return 0;
}
```

✎ *How would you implement move() and samePlayer()?*

BoardGame 1.6

The Runner now needs access to the get() and inRange() methods so we make them public:

```
public interface BoardGame {
  ...
  public char get(int col, int row)
    throws AssertionException;
  public boolean inRange(int col, int row);
  ...
}
```

➤ Which methods should be public?

✔ *Only publicize methods that clients will really need, and will not break encapsulation.*

If a client needs to be able to modify your internal state, there is something wrong with your design! (Strong coupling)

Gomoku

Gomoku is similar to TicTacToe, except it is played on a 19x19 Go board, and the winner must get 5 in a row.

```
public class Gomoku extends AbstractBoardGame {
  public Gomoku(Player playerX, Player playerO)
  {
    // 19x19 board with winning score = 5
    this.init(19,19,5,playerX, playerO);
  }
}
```

In the end, both Gomoku and TicTacToe were able to inherit everything except their constructor from AbstractGameBoard, which suggest it may really not be so abstract.

Summary

You should know the answers to these questions:

- ❑ How does polymorphism help in writing generic code?
- ❑ When should features be declared protected rather than public or private?
- ❑ How do abstract classes help to achieve code reuse?
- ❑ What is refactoring? Why should you do it in small steps?
- ❑ How do interfaces support polymorphism?

Can you answer the following questions?

- ✎ *What would change if we didn't declare AbstractBoardGame to be abstract?*
- ✎ *How does an interface (in Java) differ from a class whose methods are all abstract?*
- ✎ *Can you write generic toString() and invariant() methods for AbstractBoardGame?*
- ✎ *How could you use polymorphism (instead of a boolean flag) to make printing optional in playGame()? Does this improve the design?*
- ✎ *Is TicTacToe a special case of Gomoku, or the other way around?*
- ✎ *How would you reorganize the class hierarchy so that you could run Gomoku with boards of different sizes?*

6. Programming Tools

Overview

- ❑ Integrated Development Environments — CodeWarrior, SNiFF ...
- ❑ Debuggers
- ❑ Version control — RCS, CVS
- ❑ Profilers
- ❑ Documentation generation — Javadoc

Sources

- ❑ CodeWarrior: www.metrowerks.com
- ❑ SNiFF+: www.takefive.com

Integrated Development Environments

An Integrated Development Environment (IDE) provides a common interface to a suite of programming tools:

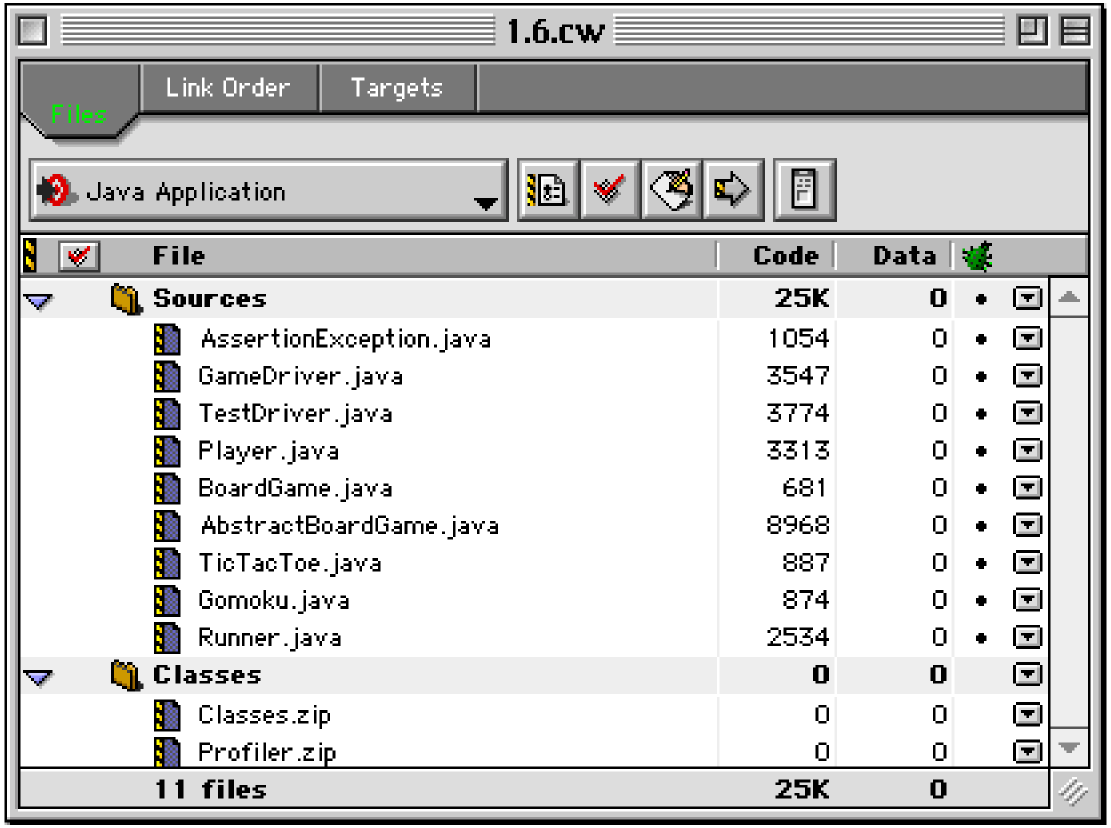
- ❑ project manager
- ❑ browsers and editors
- ❑ compilers and linkers
- ❑ make utility
- ❑ version control system
- ❑ interactive debugger
- ❑ profiler
- ❑ memory usage monitor
- ❑ documentation generator

Many of the graphical object-oriented programming tools were pioneered in Smalltalk.

CodeWarrior

CodeWarrior is a popular IDE for C, C++, Pascal and Java available for MacOS, Windows and Solaris.

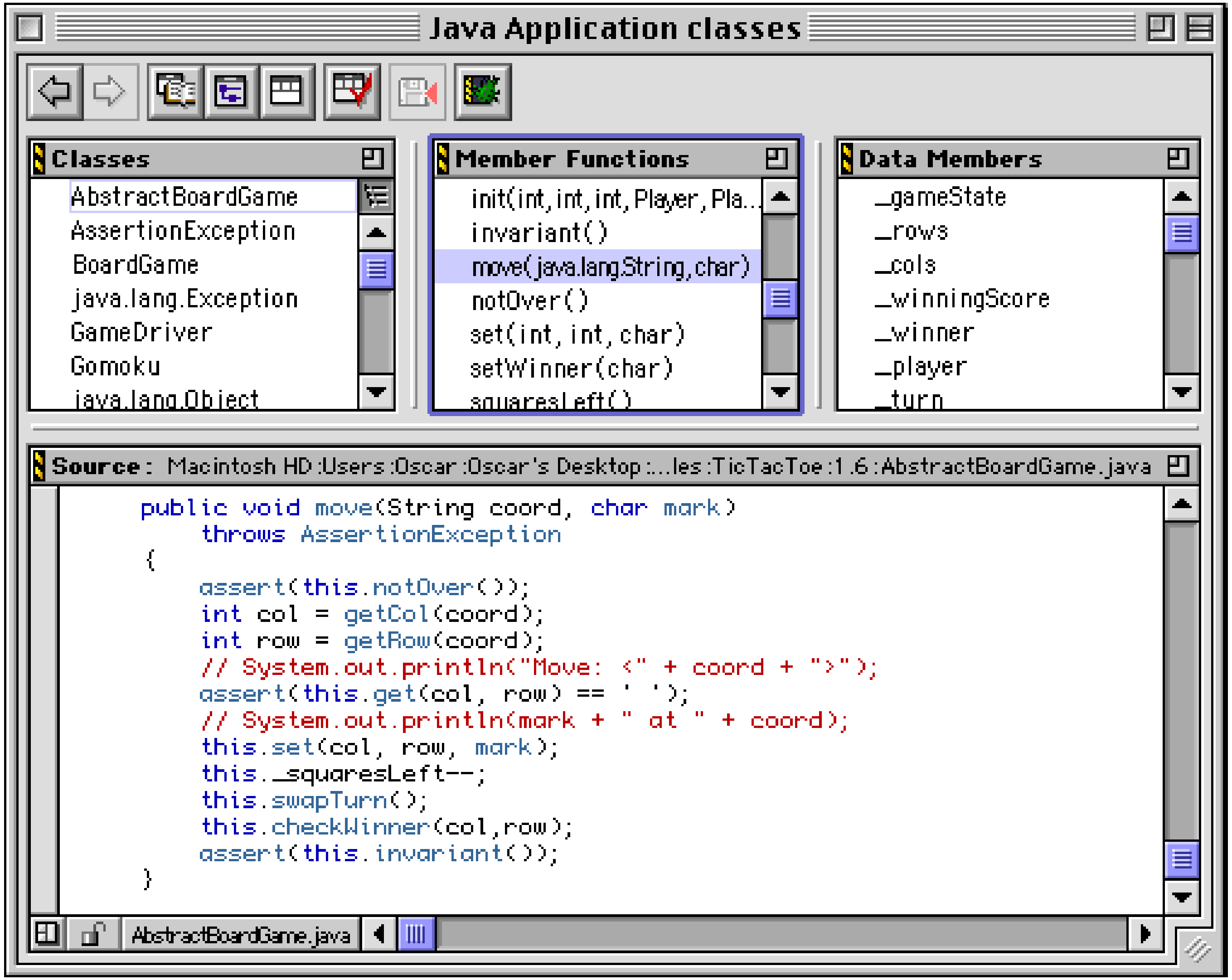
The Project Browser organizes the source and object files belonging to a project, and lets you modify the project settings, edit source files, and compile and run the application.



File	Code	Data	
Sources	**25K**	**0**	•
AssertionException.java	1054	0	•
GameDriver.java	3547	0	•
TestDriver.java	3774	0	•
Player.java	3313	0	•
BoardGame.java	681	0	•
AbstractBoardGame.java	8968	0	•
TicTacToe.java	887	0	•
Gomoku.java	874	0	•
Runner.java	2534	0	•
Classes	**0**	**0**	
Classes.zip	0	0	
Profiler.zip	0	0	
11 files	**25K**	**0**	

CodeWarrior Class Browser

The Class Browser provides one way to navigate and edit project files ...



Classes

- AbstractBoardGame
- AssertionException
- BoardGame
- java.lang.Exception
- GameDriver
- Gomoku
- java.lang.Object

Member Functions

- init(int, int, int, Player, Pla...
- invariant()
- move(java.lang.String, char)
- notOver()
- set(int, int, char)
- setWinner(char)
- squaresLeft()

Data Members

- _gameState
- _rows
- _cols
- _winningScore
- _winner
- _player
- _turn

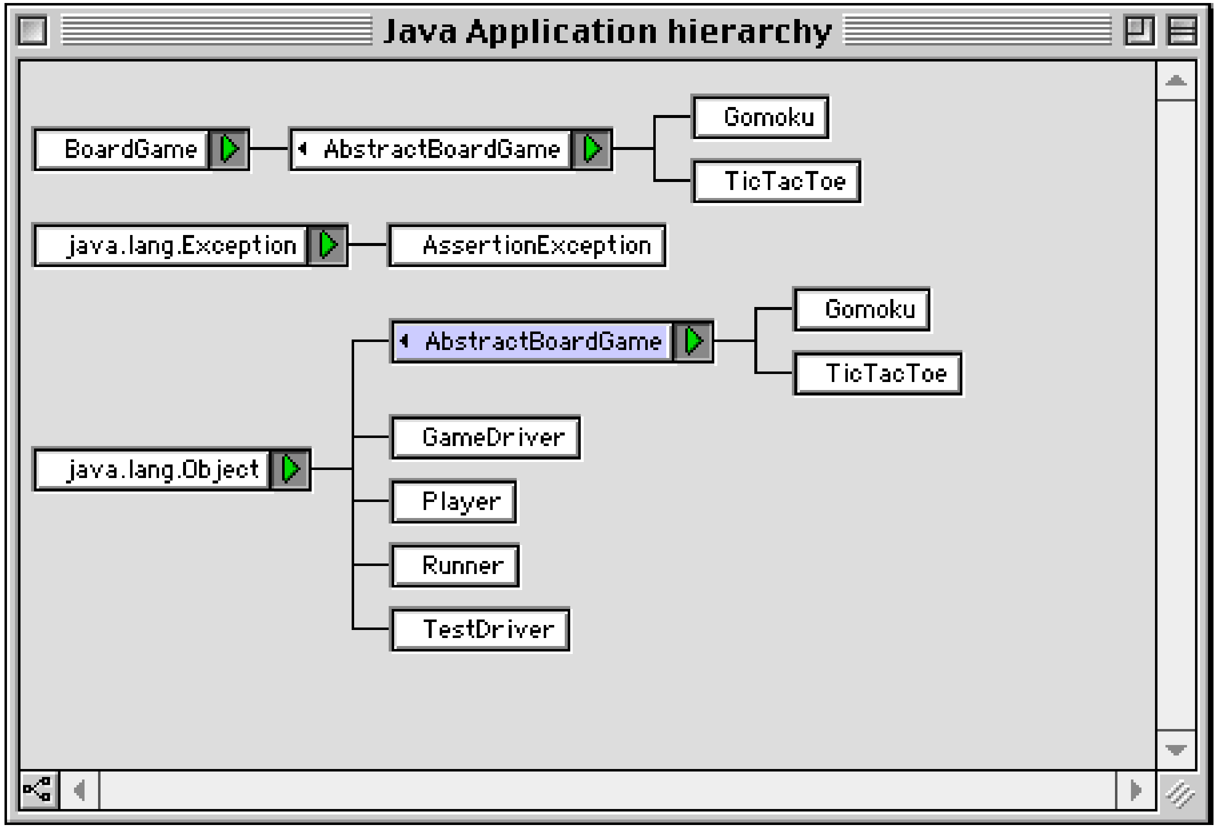
Source : Macintosh HD :Users :Oscar :Oscar's Desktop :...les :TicTacToe :1.6 :AbstractBoardGame.java

```
public void move(String coord, char mark)
    throws AssertionException
{
    assert(this.notOver());
    int col = getCol(coord);
    int row = getRow(coord);
    // System.out.println("Move: <" + coord + ">");
    assert(this.get(col, row) == ' ');
    // System.out.println(mark + " at " + coord);
    this.set(col, row, mark);
    this._squaresLeft--;
    this.swapTurn();
    this.checkWinner(col,row);
    assert(this.invariant());
}
```

CodeWarrior Hierarchy Browser

A Hierarchy Browser provides a view of the class hierarchy.

NB: no distinction is made between interfaces and classes. Classes that implement multiple interfaces appear multiple times in the hierarchy!



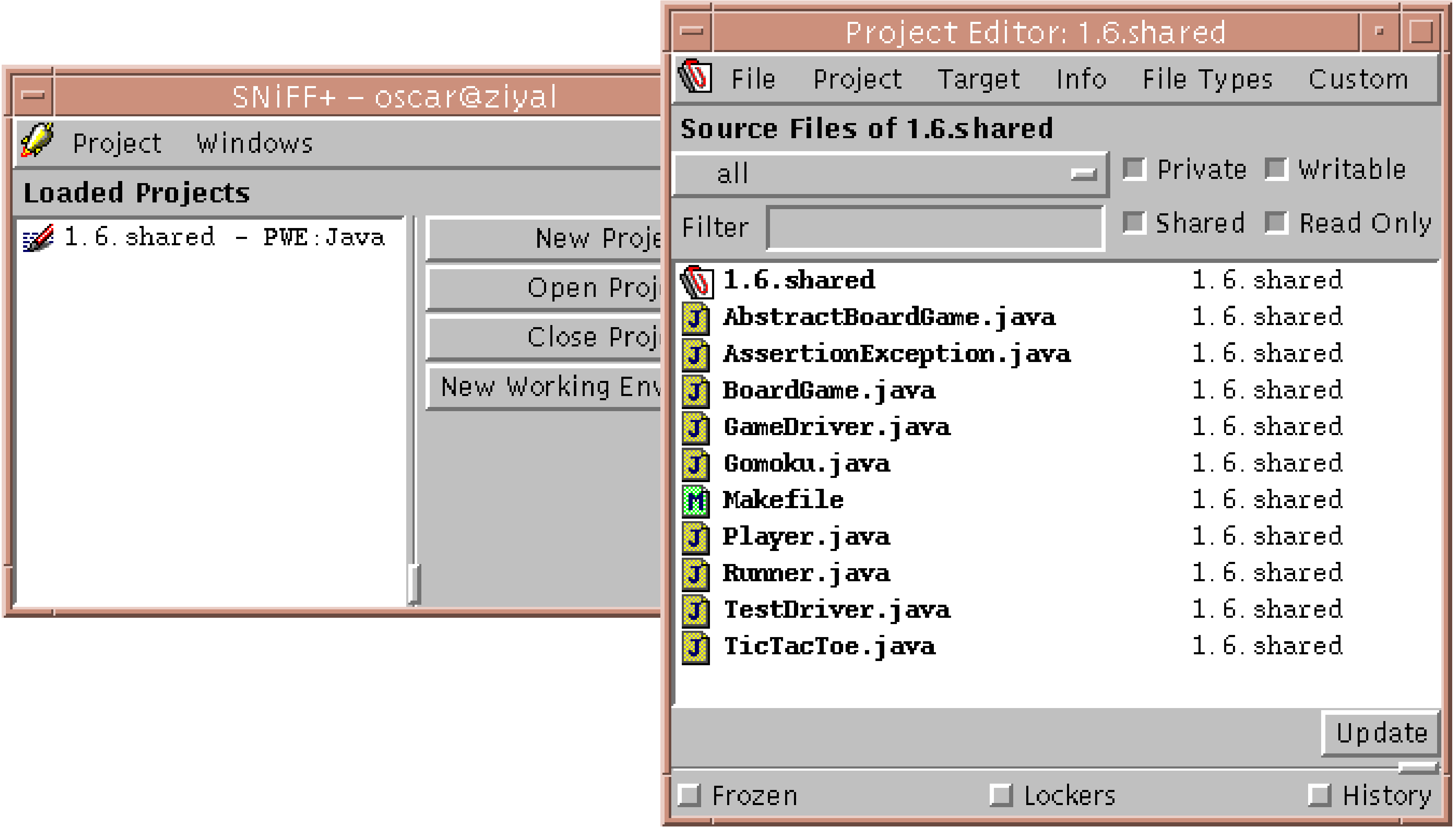
SNiFF+

SNiFF+ is an integrated development environment for C++, Java, Python and many other languages, running on Unix. It provides:

- ❑ project management
- ❑ hierarchy browser
- ❑ class browser
- ❑ symbol browser
- ❑ cross referencer
- ❑ source code editor (either built-in or external)
- ❑ version control (using RCS)
- ❑ compiler error parsing
- ❑ integrated make facility (using Unix make)

SNiFF+ is an open IDE, allowing different compilers, debuggers, etc. to be plugged in.

SNiFF+ Project Editor



Loaded Projects

1.6.shared - PWE:Java

Source Files of 1.6.shared

all

□ Private □ Writable

Filter

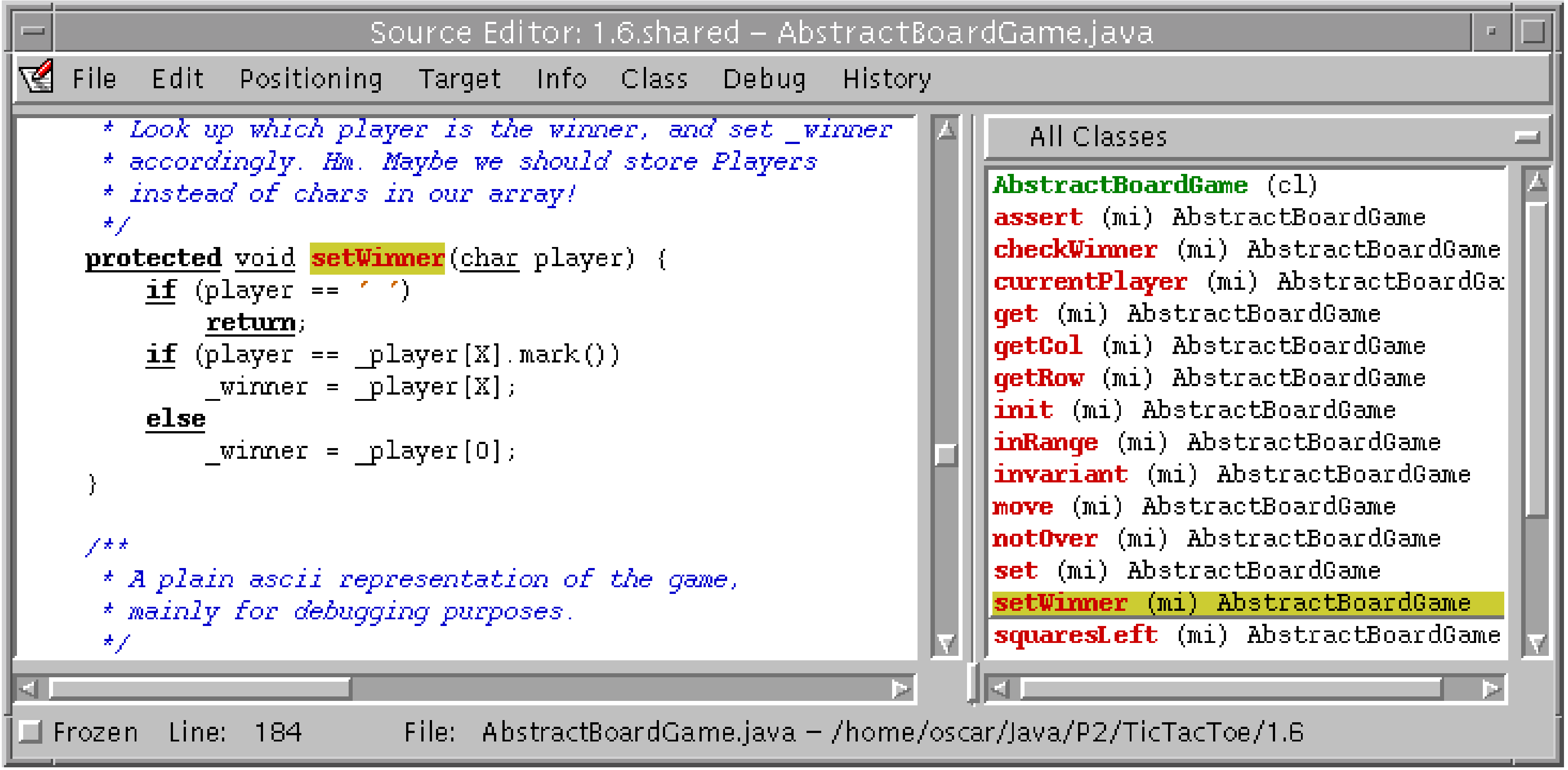
□ Shared □ Read Only

1.6.shared	1.6.shared
AbstractBoardGame.java	1.6.shared
AssertionException.java	1.6.shared
BoardGame.java	1.6.shared
GameDriver.java	1.6.shared
Gomoku.java	1.6.shared
Makefile	1.6.shared
Player.java	1.6.shared
Runner.java	1.6.shared
TestDriver.java	1.6.shared
TicTacToe.java	1.6.shared

□ Frozen □ Lockers □ History

SNiFF+ supports project development by teams: projects may be private, or shared.

SNiFF+ Source Editor

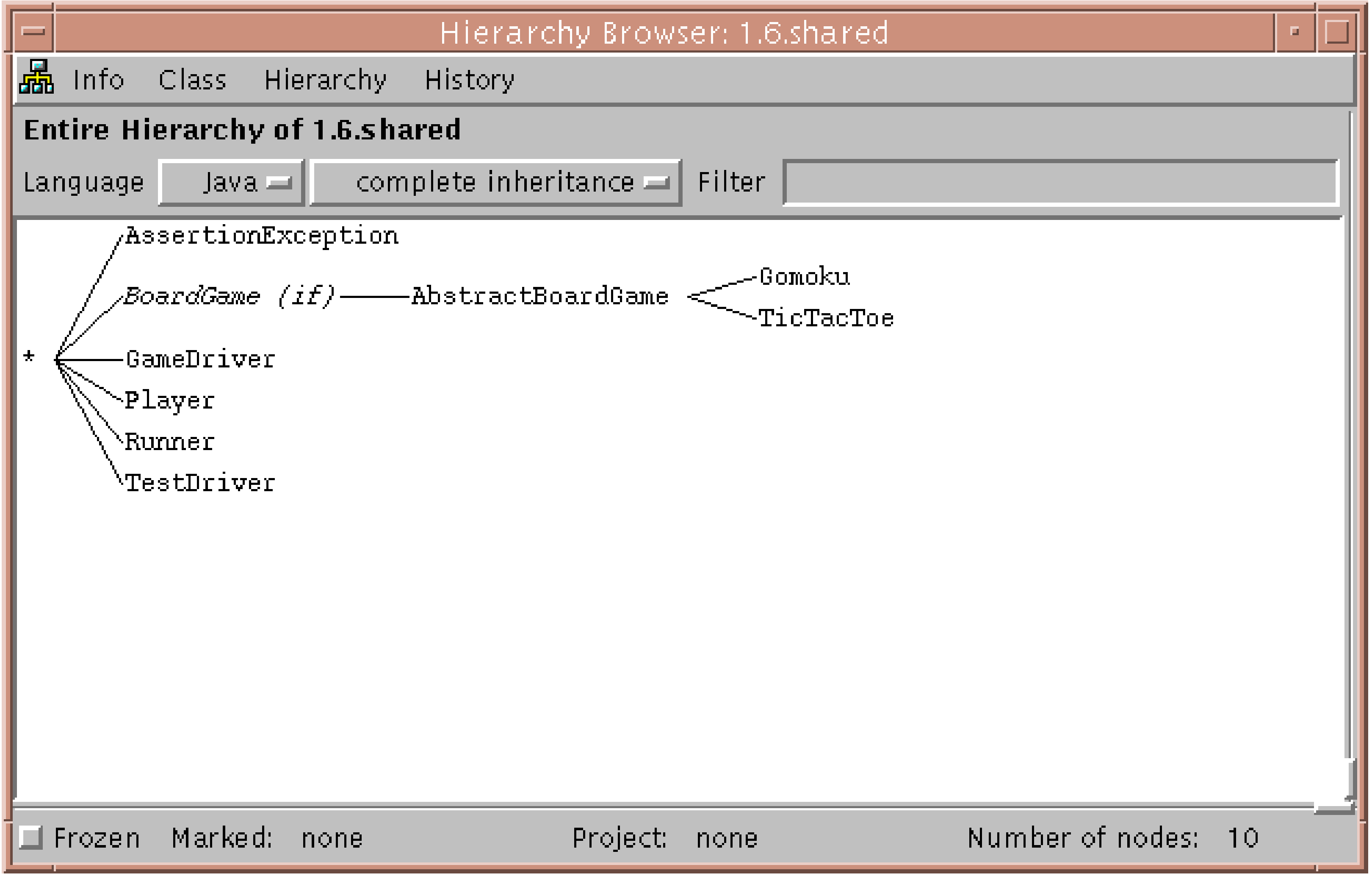


```
     * Look up which player is the winner, and set _winner
     * accordingly. Hm. Maybe we should store Players
     * instead of chars in our array!
     */
    protected void setWinner(char player) {
        if (player == ' ')
            return;
        if (player == _player[X].mark())
            _winner = _player[X];
        else
            _winner = _player[0];
    }

    /**
     * A plain ascii representation of the game,
     * mainly for debugging purposes.
     */
```

```
AbstractBoardGame (cl)
assert (mi) AbstractBoardGame
checkWinner (mi) AbstractBoardGame
currentPlayer (mi) AbstractBoardGa
get (mi) AbstractBoardGame
getCol (mi) AbstractBoardGame
getRow (mi) AbstractBoardGame
init (mi) AbstractBoardGame
inRange (mi) AbstractBoardGame
invariant (mi) AbstractBoardGame
move (mi) AbstractBoardGame
notOver (mi) AbstractBoardGame
set (mi) AbstractBoardGame
setWinner (mi) AbstractBoardGame
squaresLeft (mi) AbstractBoardGame
```

□ Frozen Line: 184 File: AbstractBoardGame.java – /home/oscar/Java/P2/TicTacToe/1.6

SNiFF+ Hierarchy Browser



Hierarchy Browser: 1.6.shared

Entire Hierarchy of 1.6.shared

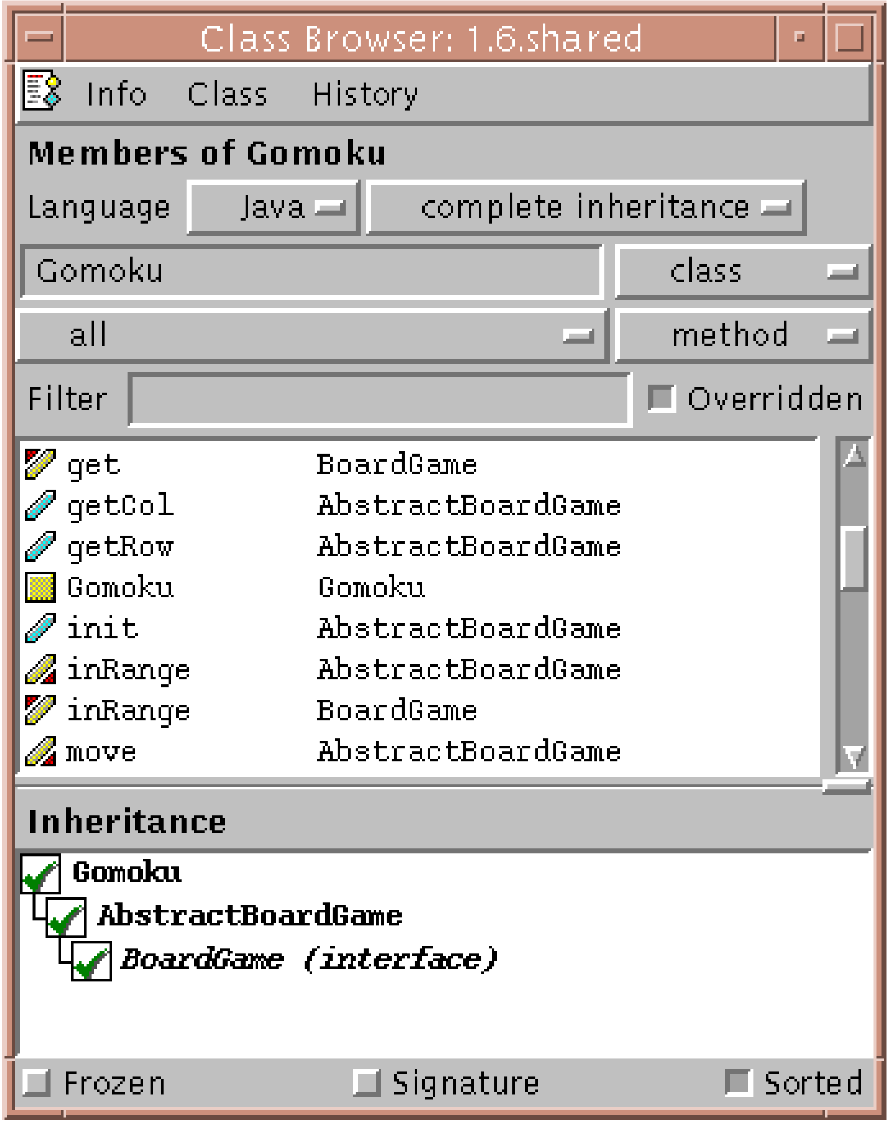
Language Java complete inheritance Filter

□ Frozen Marked: none Project: none Number of nodes: 10

SNiFF+ Class Browser

The SNiFF+ class browser shows (by the colours) which features are public, protected or private and (by the icons) which are inherited or overridden.

You can select which features you want to view (using menus, checkboxes and filters).



Members of Gomoku

Language Java complete inheritance

Gomoku class

all method

Filter □ Overridden

get	BoardGame
getCol	AbstractBoardGame
getRow	AbstractBoardGame
Gomoku	Gomoku
init	AbstractBoardGame
inRange	AbstractBoardGame
inRange	BoardGame
move	AbstractBoardGame

Inheritance

- ☑ **Gomoku**
  - ☑ **AbstractBoardGame**
    - ☑ ***BoardGame (interface)***

□ Frozen □ Signature □ Sorted

Debuggers

A debugger is a tool that allows you to examine the state of a running program:

- ❑ *step* through the program instruction by instruction
- ❑ *view* the source code of the executing program
- ❑ *inspect* (and modify) values of variables in various formats
- ❑ *set and unset breakpoints* anywhere in your program
- ❑ *execute* up to a specified breakpoint
- ❑ examine the state of an aborted program (in a “core file”)

➤ When should you use a debugger?

✔ *When you are unsure why (or where) your program is not working.*

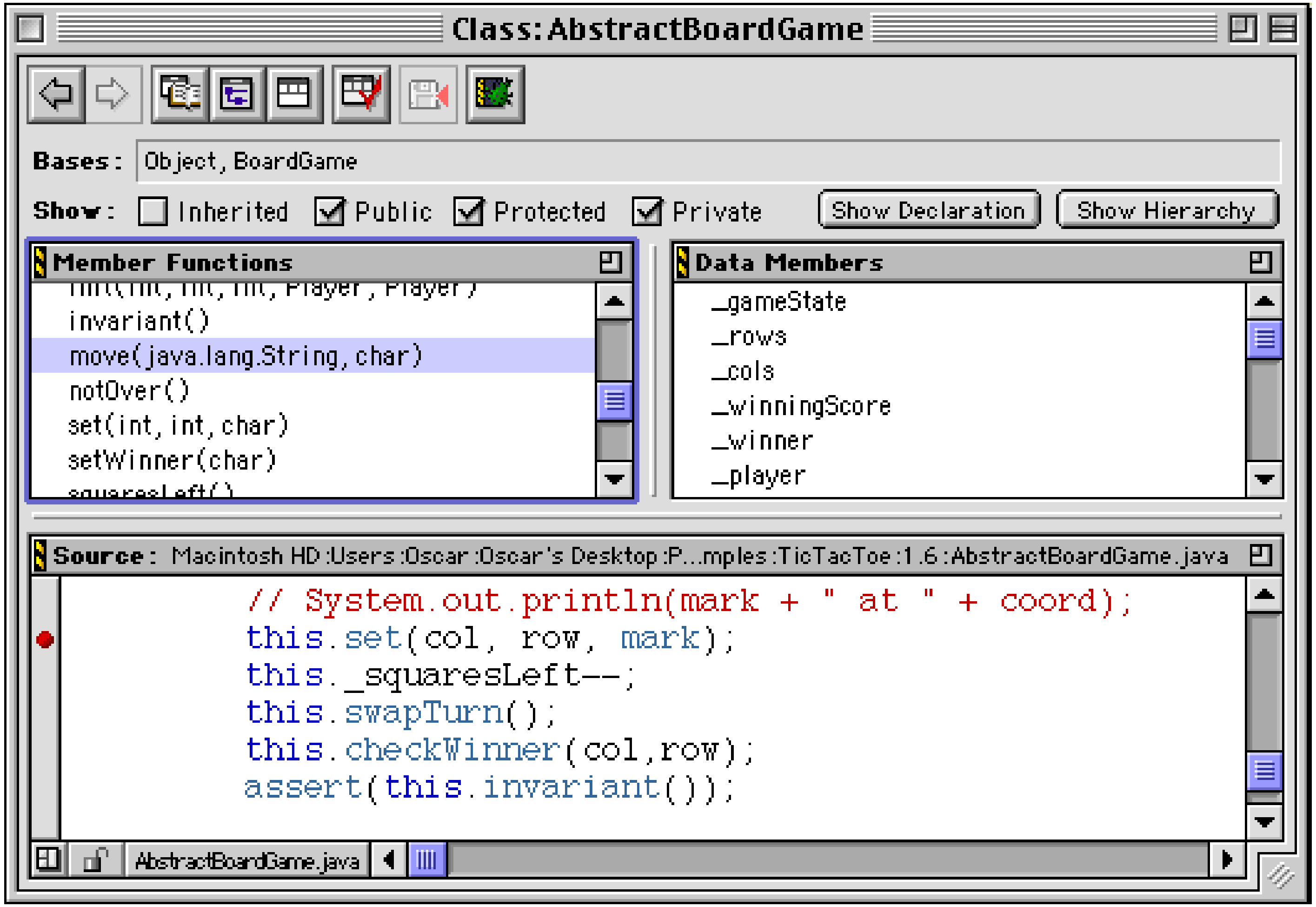
Interactive debuggers are available for most mature programming languages.

Classical debuggers are line-oriented (e.g., jdb); most modern ones are graphical.

NB: debuggers are object code specific, so can only be used with programs compiled with compilers generating compatible object files.

Setting Breakpoints

The CodeWarrior IDE lets you set breakpoints by simply clicking next to the statements where execution should be interrupted.



Class: AbstractBoardGame

Bases: Object, BoardGame

Show: □ Inherited ☑ Public ☑ Protected ☑ Private

Member Functions:

```
init(int, int, int, Player, Player)
invariant()
move(java.lang.String, char)
notOver()
set(int, int, char)
setWinner(char)
squaresLeft()
```

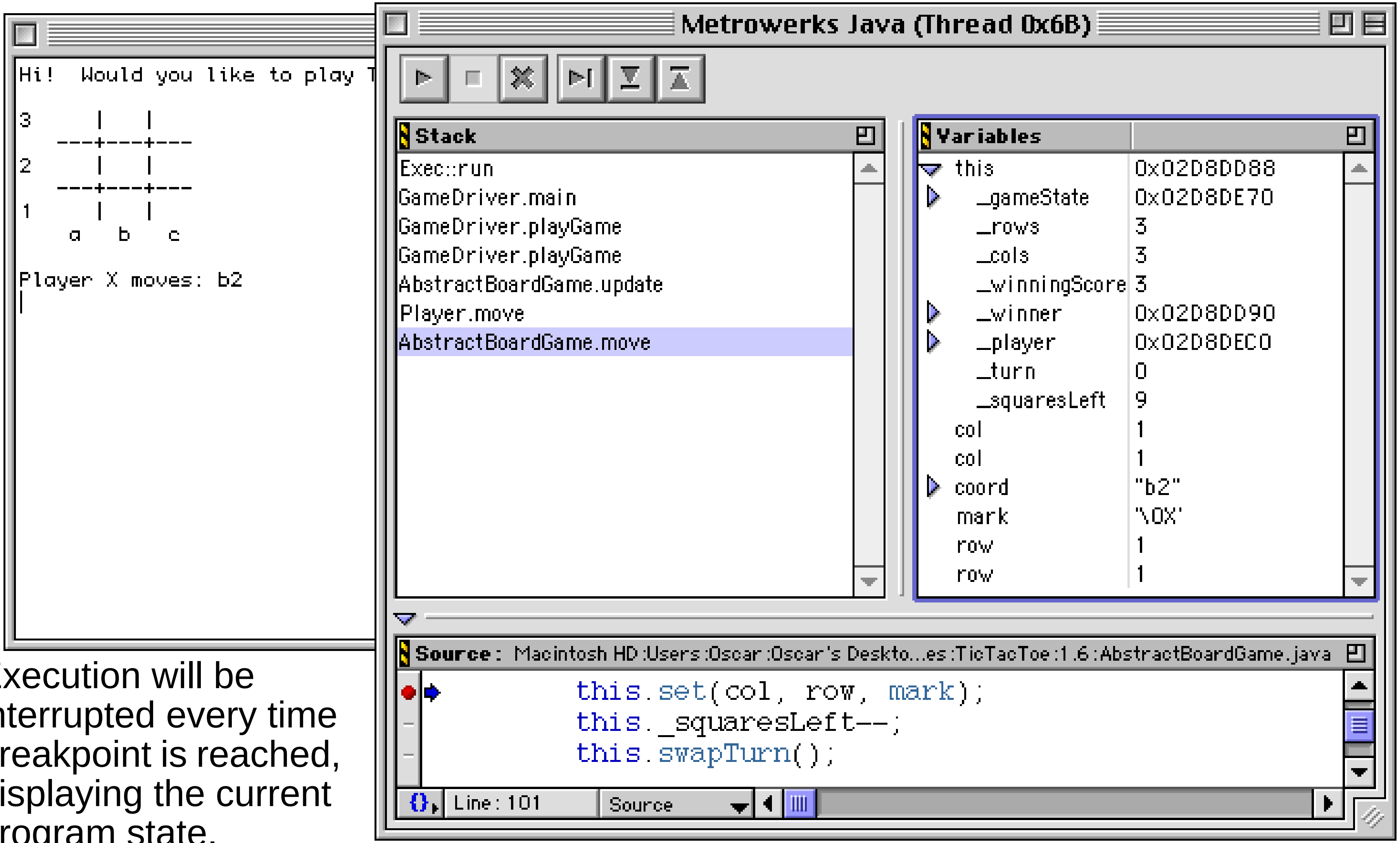
Data Members:

```
_gameState
_rows
_cols
_winningScore
_winner
_player
```

Source: Macintosh HD:Users:Oscar:Oscar's Desktop:P...mples:TicTacToe:1.6:AbstractBoardGame.java

```
// System.out.println(mark + " at " + coord);
this.set(col, row, mark);
this._squaresLeft--;
this.swapTurn();
this.checkWinner(col,row);
assert(this.invariant());
```

Debugging



```
Hi!  Would you like to play T

3     |   |
   ---+---+---
2     |   |
   ---+---+---
1     |   |
    a   b   c

Player X moves: b2
|
```

Stack

```
Exec::run
GameDriver.main
GameDriver.playGame
GameDriver.playGame
AbstractBoardGame.update
Player.move
AbstractBoardGame.move
```

Variables	
this	0x02D8DD88
_gameState	0x02D8DE70
_rows	3
_cols	3
_winningScore	3
_winner	0x02D8DD90
_player	0x02D8DEC0
_turn	0
_squaresLeft	9
col	1
col	1
coord	"b2"
mark	'\0X'
row	1
row	1

Source : Macintosh HD :Users :Oscar :Oscar's Deskto...es :TicTacToe :1.6 :AbstractBoardGame.java

```java
        this.set(col, row, mark);
        this._squaresLeft--;
        this.swapTurn();
```

Execution will be interrupted every time breakpoint is reached, displaying the current program state.

Debugging Strategy

Develop tests as you program

- ❑ Apply Design by Contract to decorate classes with invariants and pre- and post-conditions
- ❑ Develop unit tests to exercise all paths through your program
  - ☞ use assertions (not print statements) to proble the program state
  - ☞ print the state only when an assertion fails
- ❑ After every modification, do regression testing!

If errors arise during testing or usage

- ❑ Use the test results to track down and fix the bug
- ❑ If you can't tell where the bug is, then
  - ☞ use a debugger to identify the faulty code
  - ☞ fix the bug
  - ☞ identify and add any missing tests!

Version Control

A version control system keeps track of multiple file revisions:

- ❑ check-in and check-out of files
- ❑ logging changes (who, where, when)
- ❑ merge and comparison of versions
- ❑ retrieval of arbitrary versions
- ❑ "freezing" of versions as releases
- ❑ reduces storage space (manages sources files + multiple "deltas")

SCCS and RCS are two popular version control systems for UNIX.
CVS is popular on Mac, Windows and UNIX platforms (see www.cyclic.com)

➤ What kind of projects can benefit from versioning?

✔ *Use a version control system to keep track of all your projects!*

Version control is as important as testing in iterative development!

RCS

Overview of RCS commands:

❑	ci	Check in revisions
❑	co	Check out revisions
❑	rcs	Set up or change attributes of RCS files
❑	ident	Extract keyword values from an RCS file
❑	rlog	Display a summary of revisions
❑	merge	Incorporate changes from two files into a third
❑	rcsdiff	Report differences between revisions
❑	rcsmerge	Incorporate changes from two RCS files into a third
❑	rcsclean	Remove working files that have not been changed
❑	rcsfreeze	Label the files that make up a configuration

Using RCS

When `file` is checked in, an RCS file called `file,v` is created in the RCS directory:

```
mkdir RCS        # create subdirectory for RCS files
ci file          # put file under control of RCS
```

Working copies must be checked out and checked in.

```
co -l file       # check out (and lock) file for editing
ci file          # check in a modified file
co file          # check out a read-only copy
ci -u file       # check in file, but leave a read-only copy
rcsdiff file     # report changes between versions
```

Additional RCS Features

Keyword substitution

- ❑ Various keyword variables are maintained by RCS:

`$Author$`	who checked in revision (username)
`$Date$`	date and time of check-in
`$Log$`	description of revision (prompted during check-in)

and several others ...

Revision numbering:

- ❑ Usually each revision is numbered *release.level*
- ❑ Level is incremented upon each check-in
- ❑ A new release is created explicitly:

```
ci -r2.0 file
```

Profilers

A profiler (e.g., java -prof) tells you where an executed program has spent its time

1. your program must first be *instrumented* by (i) setting a compiler (or interpreter) option, or (ii) adding instrumentation code to your source program
2. the program is run, generating a *profile data file*
3. the *profiler* is executed with the profile data as input

The profiler can then display the *call graph* in various formats ...

➤ When should you use a profiler?

✔ *Always run a profiler before attempting to tune performance.*

➤ How early should you start worrying about performance?

✔ *Only after you have a clean, running program with poor performance.*

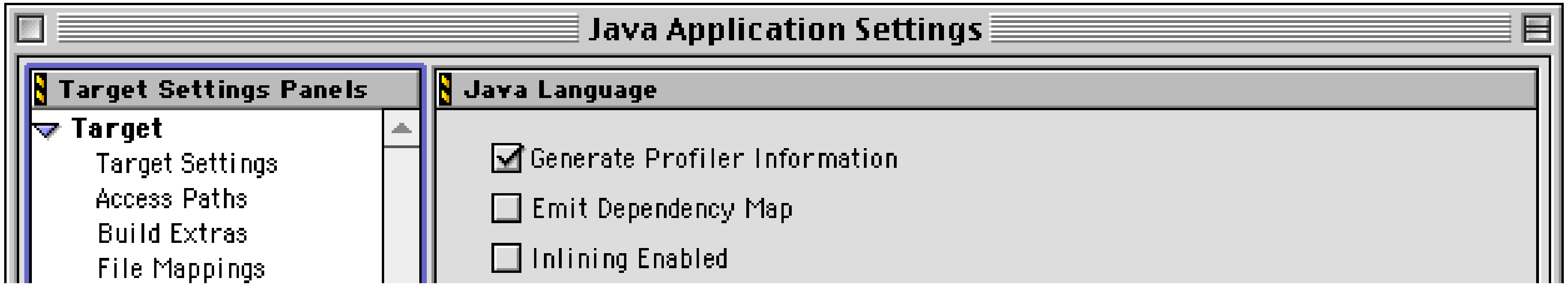
NB: The call graph also tells you which parts of the program have (not) been tested!

Profiling with CodeWarrior

Instrument the code:

```
import com.mw.Profiler.Profiler;
public class TestDriver {
   public static void main(String args[]) {
      Profiler.Init(500, 20);    // #methods; stack depth
      Profiler.StartProfiling();
      doTicTacToeTests();
      doGomokuTests();
      Profiler.StopProfiling();
      Profiler.Dump("TicTacToe Profile");
      Profiler.Terminate();
   } ...
```

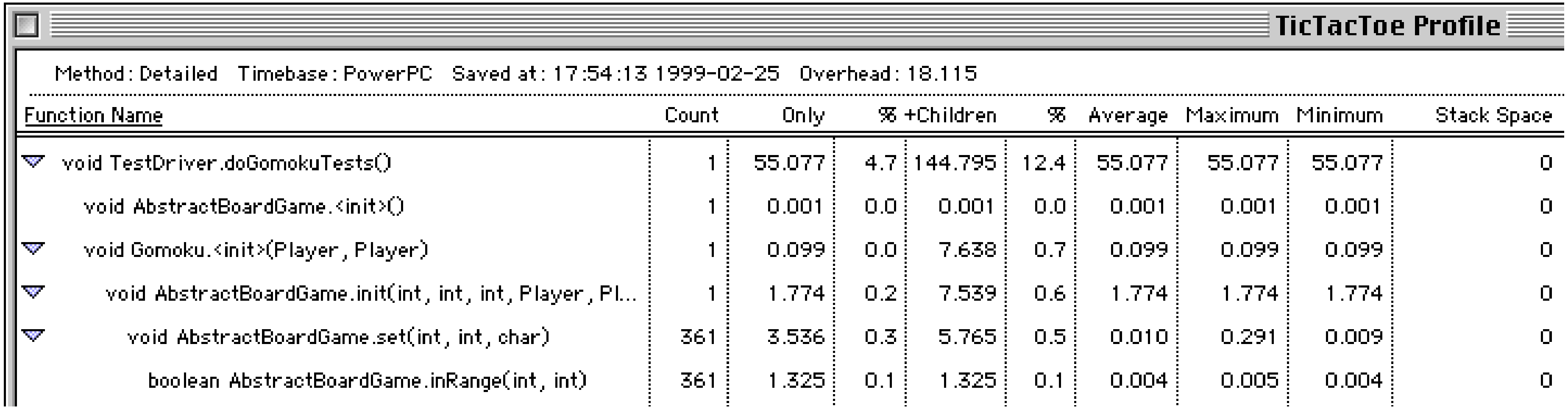
and turn on profiling:



☑ Generate Profiler Information
□ Emit Dependency Map
□ Inlining Enabled

Profile Data

Call graphs can typically be displayed hierarchically:

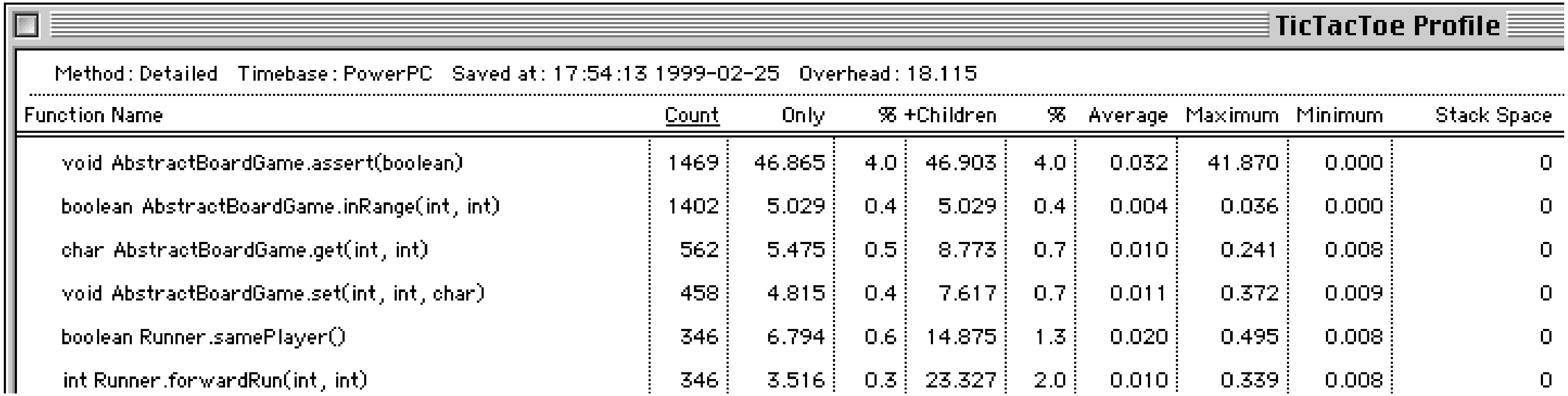


TicTacToe Profile

Method: Detailed   Timebase: PowerPC   Saved at: 17:54:13 1999-02-25   Overhead: 18.115

	Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum	Stack Space
▽	void TestDriver.doGomokuTests()	1	55.077	4.7	144.795	12.4	55.077	55.077	55.077	0
	void AbstractBoardGame.<init>()	1	0.001	0.0	0.001	0.0	0.001	0.001	0.001	0
▽	void Gomoku.<init>(Player, Player)	1	0.099	0.0	7.638	0.7	0.099	0.099	0.099	0
▽	void AbstractBoardGame.init(int, int, int, Player, Pl…	1	1.774	0.2	7.539	0.6	1.774	1.774	1.774	0
▽	void AbstractBoardGame.set(int, int, char)	361	3.536	0.3	5.765	0.5	0.010	0.291	0.009	0
	boolean AbstractBoardGame.inRange(int, int)	361	1.325	0.1	1.325	0.1	0.004	0.005	0.004	0

or sorted by timings, number of calls etc.:



TicTacToe Profile

Method: Detailed   Timebase: PowerPC   Saved at: 17:54:13 1999-02-25   Overhead: 18.115

Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum	Stack Space
void AbstractBoardGame.assert(boolean)	1469	46.865	4.0	46.903	4.0	0.032	41.870	0.000	0
boolean AbstractBoardGame.inRange(int, int)	1402	5.029	0.4	5.029	0.4	0.004	0.036	0.000	0
char AbstractBoardGame.get(int, int)	562	5.475	0.5	8.773	0.7	0.010	0.241	0.008	0
void AbstractBoardGame.set(int, int, char)	458	4.815	0.4	7.617	0.7	0.011	0.372	0.009	0
boolean Runner.samePlayer()	346	6.794	0.6	14.875	1.3	0.020	0.495	0.008	0
int Runner.forwardRun(int, int)	346	3.516	0.3	23.327	2.0	0.010	0.339	0.008	0

Javadoc

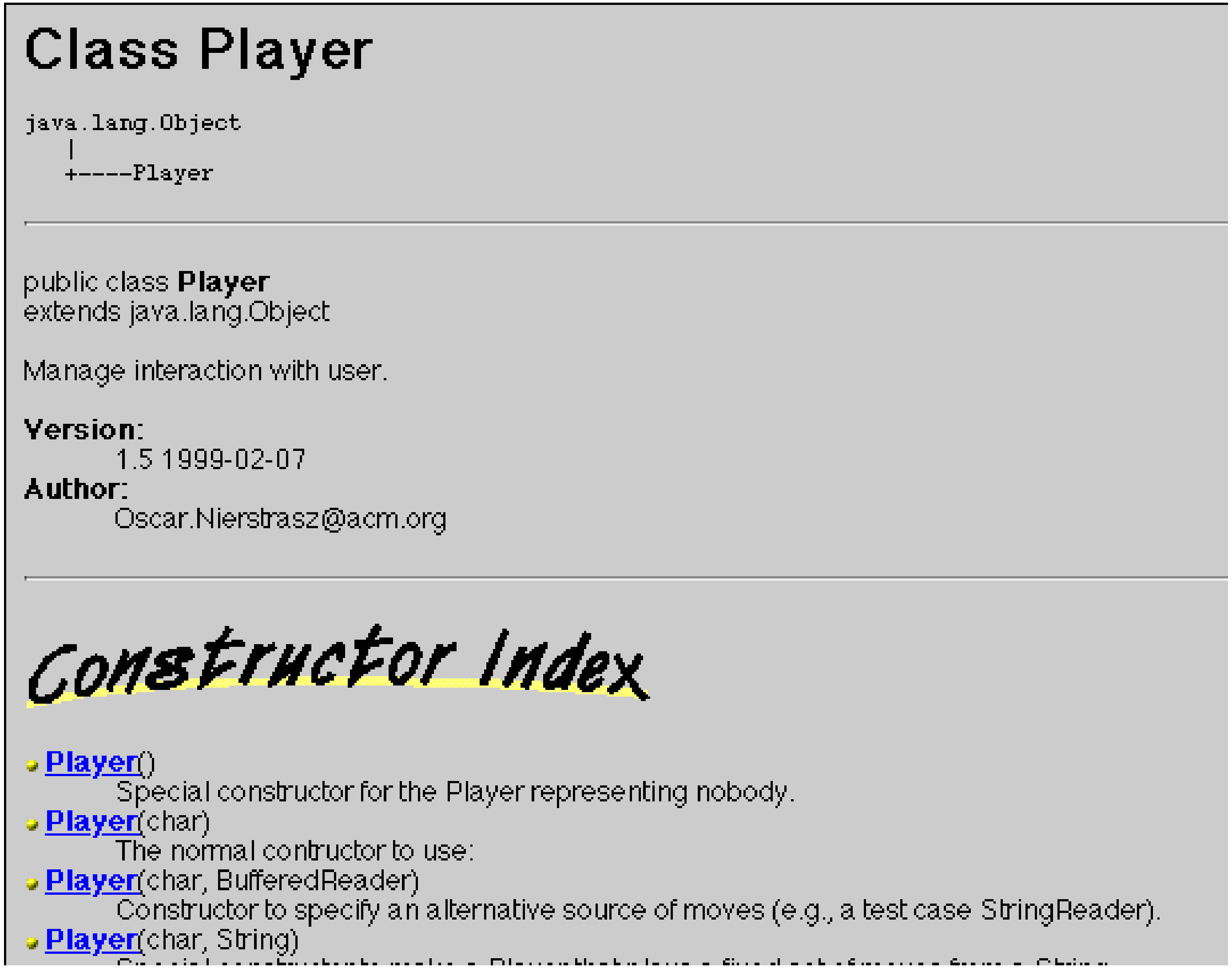
Javadoc generates API documentation in HTML format for specified Java source files.

Each *class*, *interface* and each *public* or *protected method* may be preceded by “javadoc comments” between `/**` and `*/`. Comments may contain special tag values (e.g., ...) and (some) HTML tags.

```
import java.io.*;
/**
 * Manage interaction with user.
 * @author Oscar.Nierstrasz@acm.org
 * @version 1.5 1999-02-07
 */
public class Player { ...
  /**
   * Constructor to specify an alternative source of moves
   * (e.g., a test case StringReader).
   */
  public Player(char mark, BufferedReader in) { ...
```

Javadoc output

View it with your favourite web browser!



Class Player

```
java.lang.Object
   |
   +----Player
```

public class **Player**
extends java.lang.Object

Manage interaction with user.

Version:
1.5 1999-02-07
Author:
Oscar.Nierstrasz@acm.org

Constructor Index

- **Player**()
  Special constructor for the Player representing nobody.
- **Player**(char)
  The normal contructor to use:
- **Player**(char, BufferedReader)
  Constructor to specify an alternative source of moves (e.g., a test case StringReader).
- **Player**(char, String)

Other tools

Be familiar with the programming tools in your environment!

Multi-platform tools:

- ❑ **zip/jar:** store and compress files and directories into a single “zip file”
- ❑ **memory inspection tools:** like ZoneRanger and Purify, help to detect other memory management problems, such as “memory leaks”

Unix tools:

- ❑ **make**: regenerate (compile) files when files they depend on are modified
- ❑ **diff** and **patch**: compare versions of files, and generate/apply deltas
- ❑ **awk**, **sed** and **perl**: process text files according to editing scripts/programs
- ❑ **lex** and **yacc** [flex and bison]: generate lexical analysers and parsers from regular expression and context-free grammar specification files
- ❑ **lint**: detect bugs, portability problems and other possible errors in C programs
- ❑ **strip**: remove symbol table and other non-essential data from object files

Many tools have their equivalents on other platforms ...

Summary

You should know the answers to these questions:

- ❑ When should you use a debugger?
- ❑ What are breakpoints? Where should you set them?
- ❑ What should you do after you have fixed a bug?
- ❑ What functionality does a version control system support?
- ❑ When should you use a profiler?

Can you answer the following questions?

- ✎ *How can you tell when there is a bug in the compiler (rather than in your program)?*
- ✎ *How often should you checkpoint a version of your system?*
- ✎ *When should you specify a version of your project as a new “release”?*
- ✎ *How can you tell if you have tested every part of your system*

7. A Testing Framework

Overview

- ❑ What is a framework?
- ❑ JUnit — a simple testing framework
- ❑ Money and MoneyBag — a testing case study
- ❑ Double Dispatch — how to add different types of objects
- ❑ Testing practices

Sources

- ❑ JUnit 2.1
- ❑ “Test Infected: Programmers Love Writing Tests,” Kent Beck, Erich Gamma
- ❑ “Simple Smalltalk Testing: With Patterns”, Kent Beck

All available from: ftp://www.armaties.com/

The Problem

> *“Testing is not closely integrated with development. This prevents you from measuring the progress of development — you can't tell when something starts working or when something stops working.”*

Interactive testing is *tedious* and *seldom exhaustive*.

Automated tests are better, but,

- ❑ how to introduce tests *interactively*?
- ❑ how to organize *suites* of tests?

Testing Practices

During Development

- ❑ When you need to add new functionality, write the tests first. You will be done when the test runs.
- ❑ When you need to redesign your software to add new features, refactor in steps, and run the (regression) tests after each step. Fix what's broken before proceeding.

During Debugging

- ❑ When someone discovers a defect in your code, first write a test that will succeed if the code is working. Then debug until the test succeeds.

> *"Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead."*
>
> *Martin Fowler*

JUnit

JUnit is a simple “testing framework” that provides:

- ❑ classes for writing Test *Cases* and Test *Suites*
- ❑ methods for *setting up* and *cleaning up* test data (“fixtures”)
- ❑ methods for making *assertions*
- ❑ textual and graphical tools for *running tests*

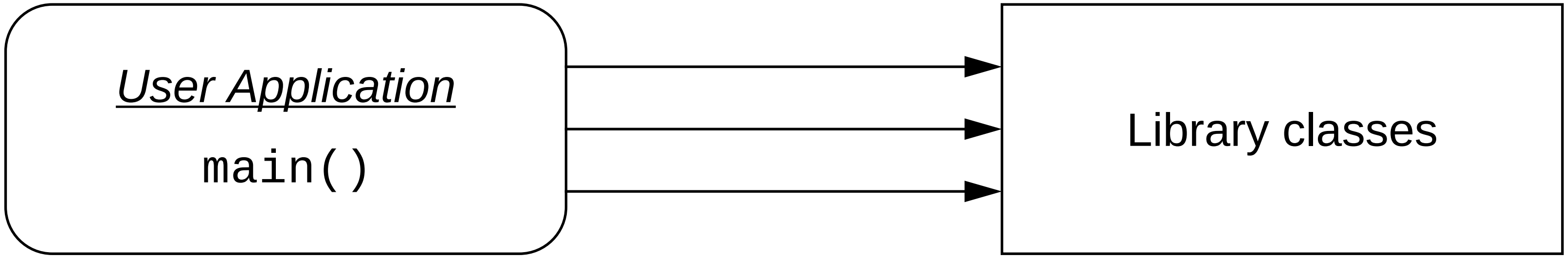
JUnit distinguishes between *failures* and *errors:*

- ❑ A *failure* is a failed assertion, i.e., an anticipated problem that you test.
- ❑ An *error* is a condition you didn’t check for.

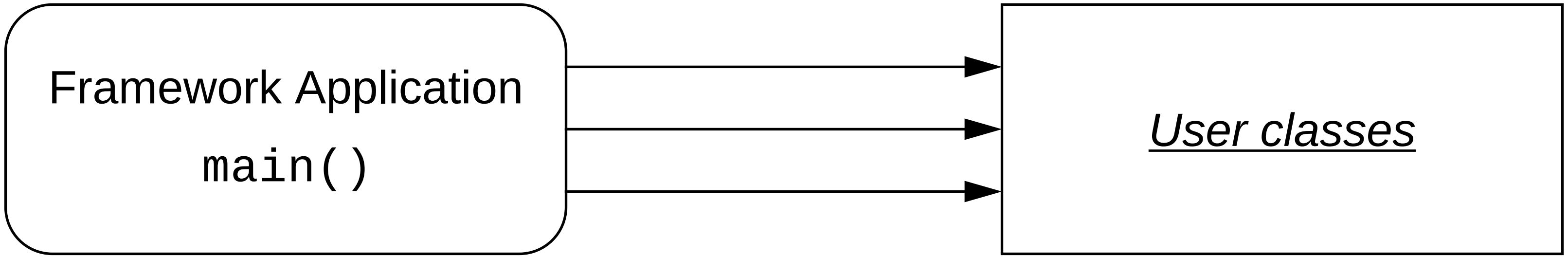
NB: this is not the same distinction made by Meyer!

Frameworks vs. Libraries

In traditional application architectures, user applications make use of library functionality in the form of procedures or classes:



A framework reverses the usual relationship between generic and application code. Frameworks provide *both* generic functionality *and* application architecture:



Essentially, a framework says: “Don’t call me — I’ll call you.”

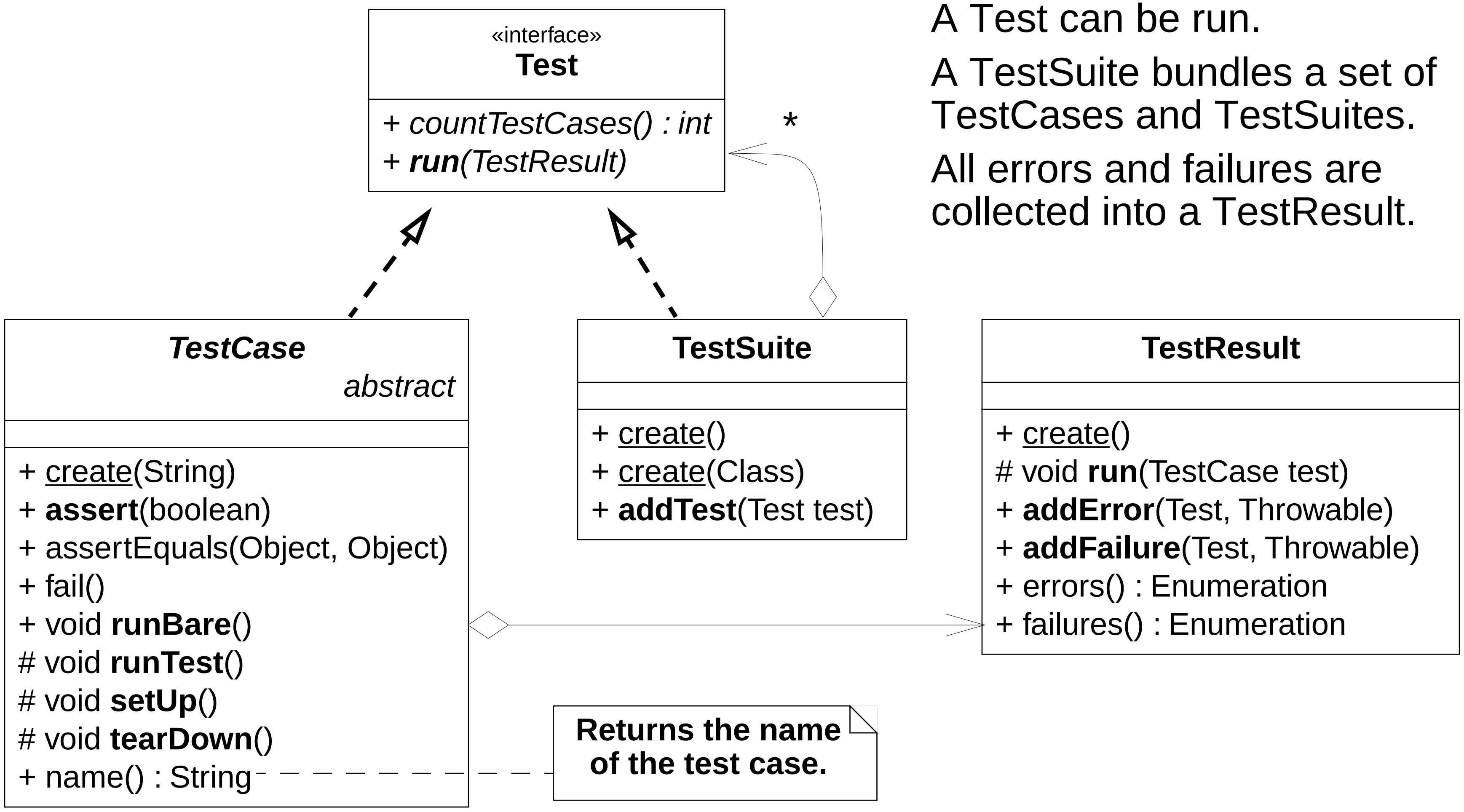
The JUnit Framework

These are the most important classes of the framework ...

A Test can be run.

A TestSuite bundles a set of TestCases and TestSuites.

All errors and failures are collected into a TestResult.



«interface»
Test

+ *countTestCases() : int*
+ ***run****(TestResult)*

*

TestCase
abstract

+ create(String)
+ **assert**(boolean)
+ assertEquals(Object, Object)
+ fail()
+ void **runBare**()
# void **runTest**()
# void **setUp**()
# void **tearDown**()
+ name() : String

TestSuite

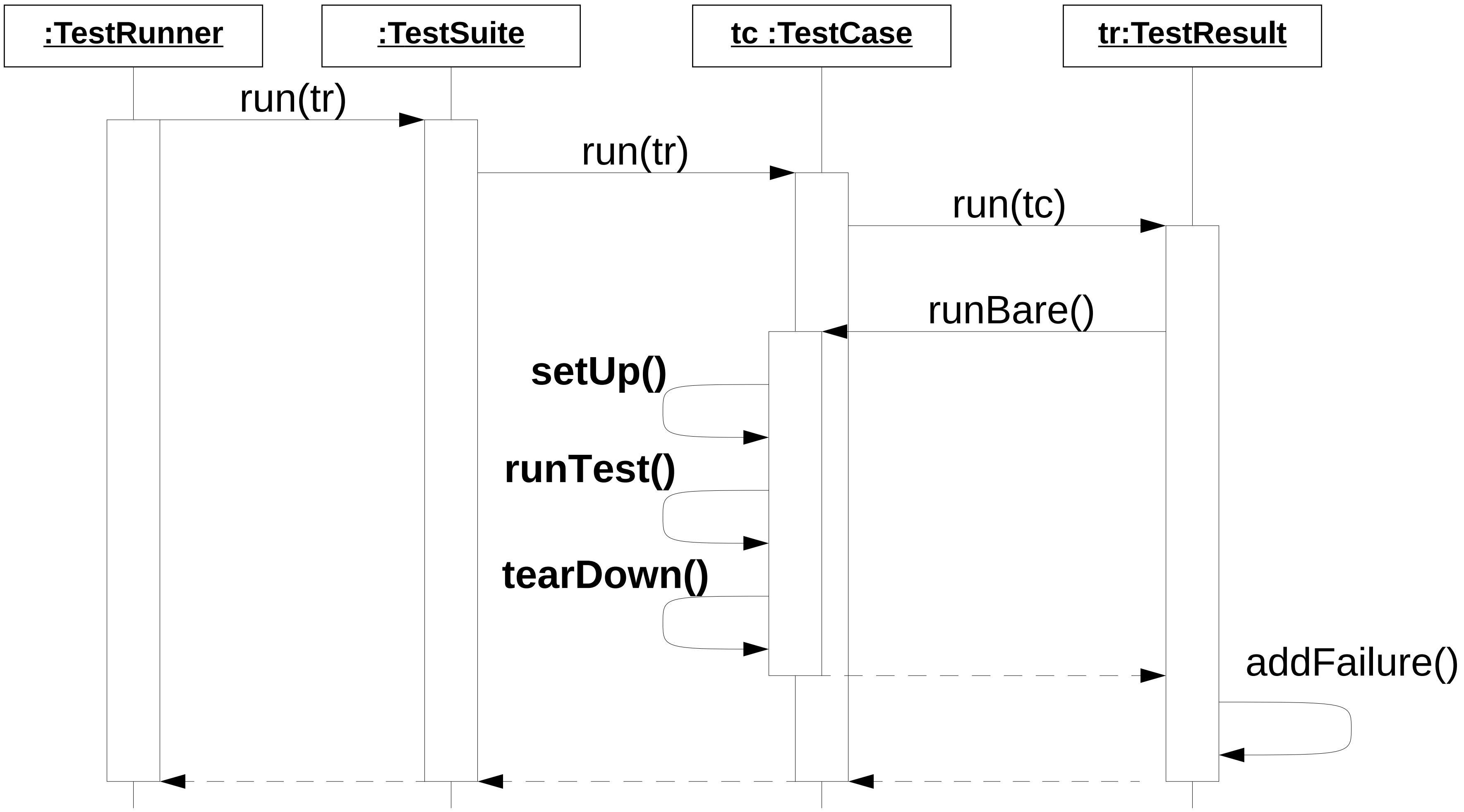
+ create()
+ create(Class)
+ **addTest**(Test test)

TestResult

+ create()
# void **run**(TestCase test)
+ **addError**(Test, Throwable)
+ **addFailure**(Test, Throwable)
+ errors() : Enumeration
+ failures() : Enumeration

Returns the name of the test case.

A Testing Scenario



The framework calls the test methods that *you define* for your test cases.

Testing Style

“The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.”

- ❑ write *unit tests* that thoroughly test a single class
- ❑ write tests *as you develop* (even *before* you implement)
- ❑ write tests for *every new piece of functionality*

“Developers should spend 25-50% of their time developing tests.”

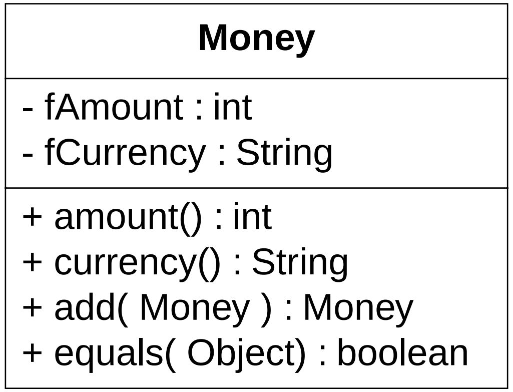
Representing multiple currencies

The problem ...

> *“The program we write will solve the problem of representing arithmetic with multiple currencies. Arithmetic between single currencies is trivial, you can just add the two amounts. ... Things get more interesting once multiple currencies are involved.”*

Money

We start by designing a *simple* Money class to handle a *single* currency:



Money

- fAmount : int
- fCurrency : String

+ amount() : int
+ currency() : String
+ add(Money) : Money
+ equals(Object) : boolean

```
class Money {
  ...
  public Money add(Money m) {
    return new Money(
          amount()+m.amount(),
          currency());
  }
  ...
}
```

NB: The first version does not consider how to add different currencies!

MoneyTest

To test our Money class, we define a *TestCase* that exercises some test data:

```
import junit.framework.*;

public class MoneyTest extends TestCase {
   private Money f12CHF;
   private Money f14CHF;

   public MoneyTest(String name) { super(name); }

   protected void setUp() {
      f12CHF = new Money(12, "CHF"); // some test data
      f14CHF = new Money(14, "CHF");
   }
   ...
}
```

Some basic tests

We define methods to test the most basic things we expect to hold ...

```
public void testEquals() {
   assert(!f12CHF.equals(null));
   assertEquals(f12CHF, f12CHF);
   assertEquals(f12CHF, new Money(12, "CHF"));
   assert(!f12CHF.equals(f14CHF));
}

public void testSimpleAdd() {
   Money expected = new Money(26, "CHF");
   Money result = f12CHF.add(f14CHF);
   assert(expected.equals(result));
   // or assertEquals(expected, result);
}
```

Building a Test Suite

... and we bundle these tests into a Test Suite:

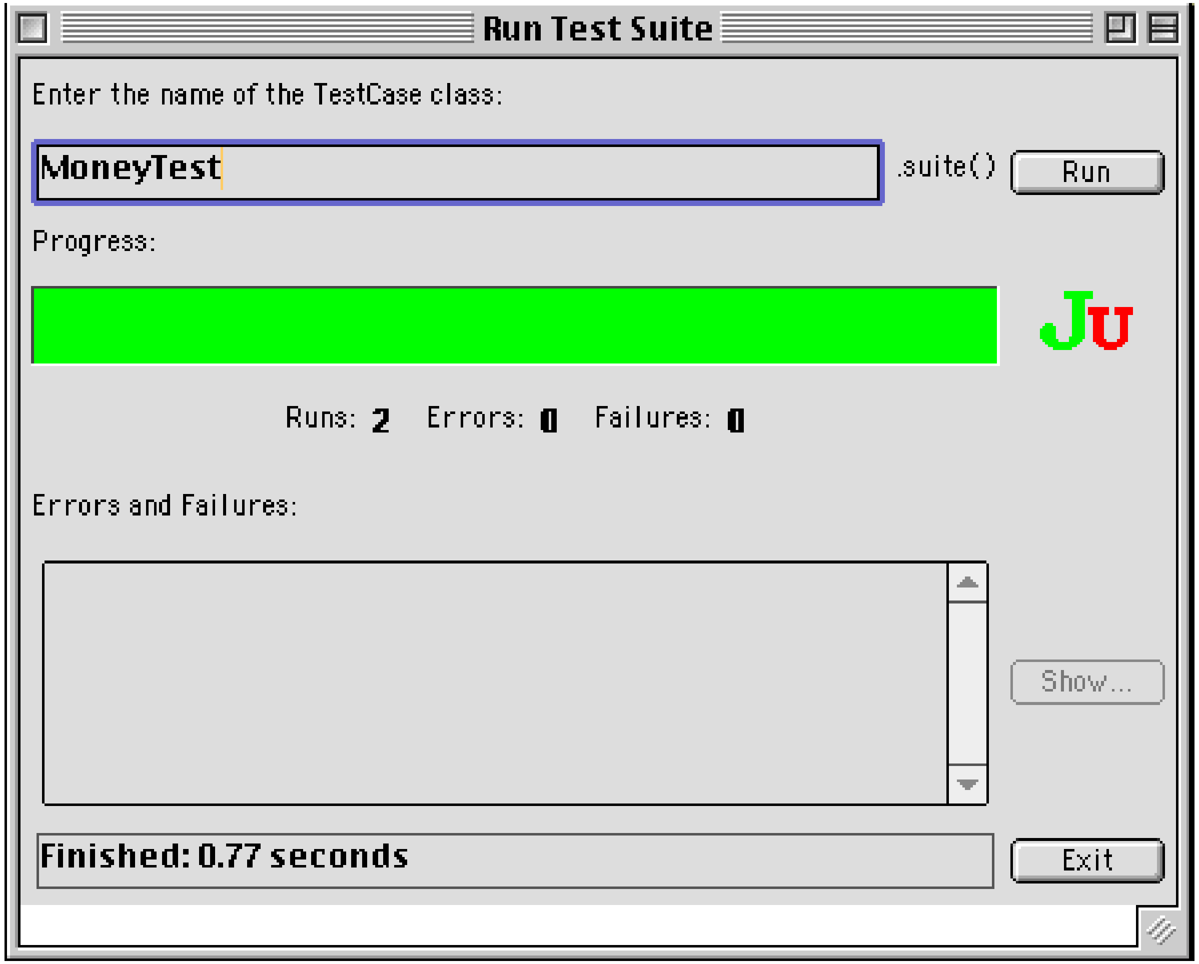
```
public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(new MoneyTest("testEquals"));
  suite.addTest(new MoneyTest("testSimpleAdd"));
  return suite;
}
```

A Test Suite:

- ❑ bundles together a bunch of named TestCase instances
- ❑ by convention, is returned by a static method called suite()

The TestRunner

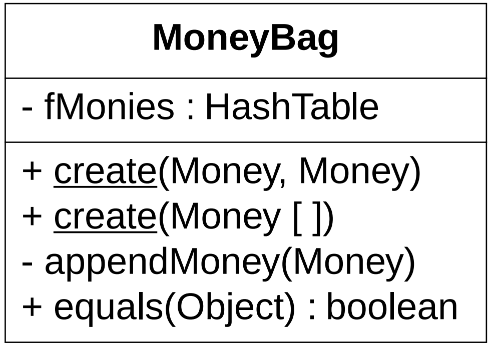
junit.ui.TestRunner is a GUI that we can use to instantiate and run the suite:



MoneyBags

To handle multiple currencies, we introduce a MoneyBag class that can hold two or more instances of Money:

```
class MoneyBag {
  ...
  MoneyBag(Money bag[]) {
    for (int i= 0; i < bag.length; i++)
      appendMoney(bag[i]);
  }
  private void appendMoney(Money aMoney) {
    Money m = (Money)
      fMonies.get(aMoney.currency());
    if (m != null)
      m = m.add(aMoney);
    else
      m = aMoney;
    fMonies.put(aMoney.currency(), m);
  }
}
```



MoneyBag

- fMonies : HashTable

+ create(Money, Money)
+ create(Money [ ])
- appendMoney(Money)
+ equals(Object) : boolean

Testing MoneyBags (I)

To test MoneyBags, we need to extend the fixture ...

```
public class MoneyTest extends TestCase {
    ...
    protected void setUp() {
        f12CHF = new Money(12, "CHF");
        f14CHF = new Money(14, "CHF");
        f7USD = new Money( 7, "USD");
        f21USD = new Money(21, "USD");
        fMB1 = new MoneyBag(f12CHF, f7USD);
        fMB2 = new MoneyBag(f14CHF, f21USD);
    }
```

Testing MoneyBags (II)

... define some new (obvious) tests ...

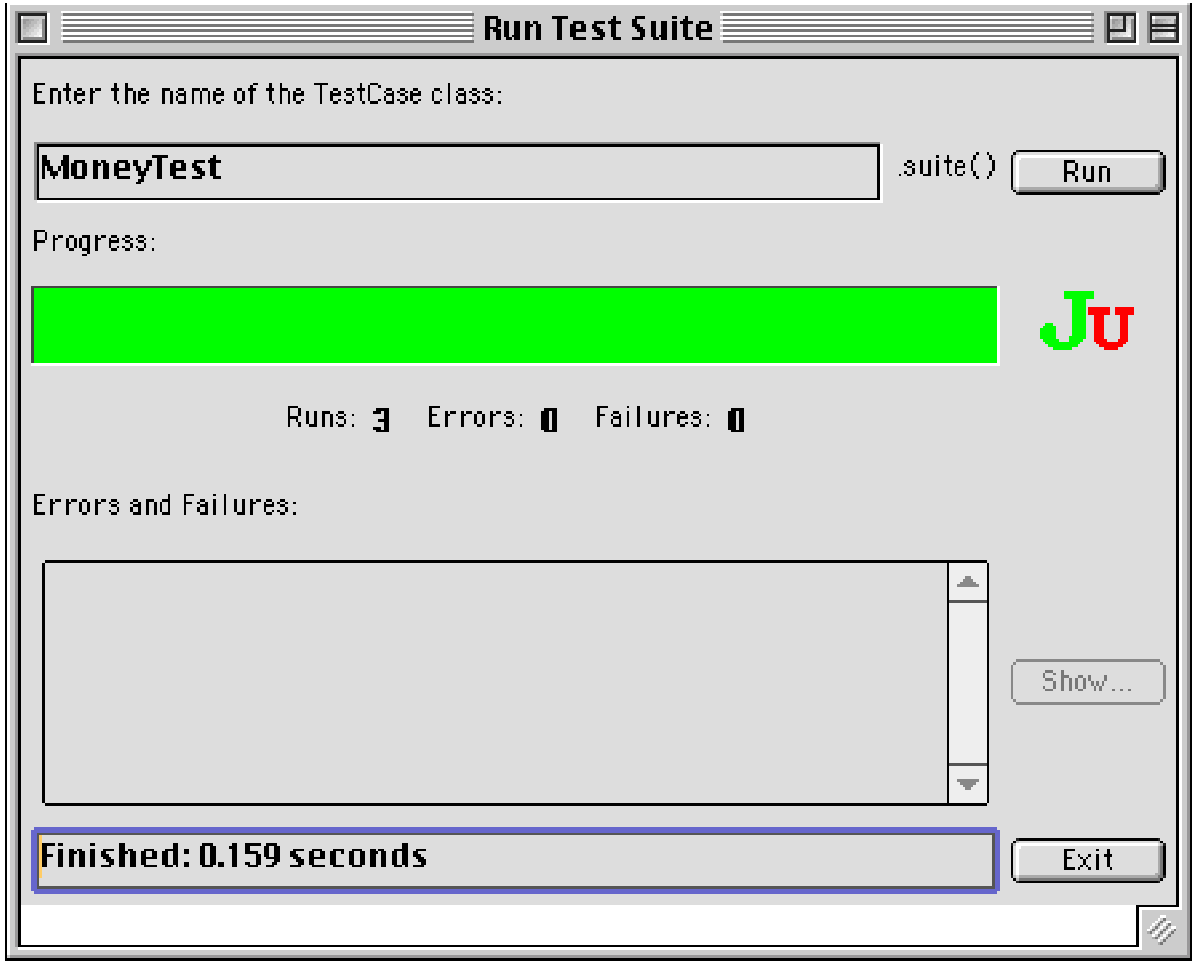
```
public void testBagEquals() {
  assert(!fMB1.equals(null));
  assertEquals(fMB1, fMB1);
  assert(!fMB1.equals(f12CHF));
  assert(!f12CHF.equals(fMB1));
  assert(!fMB1.equals(fMB2));
}
```

... add them to the test suite ...

```
public static Test suite() {
  ...
  suite.addTest(new MoneyTest("testBagEquals"));
  return suite;
}
```

Testing MoneyBags (III)

and run the tests.



Run Test Suite

Enter the name of the TestCase class:

MoneyTest .suite()

Progress:

Runs: 3 Errors: 0 Failures: 0

Errors and Failures:

Finished: 0.159 seconds

Adding MoneyBags

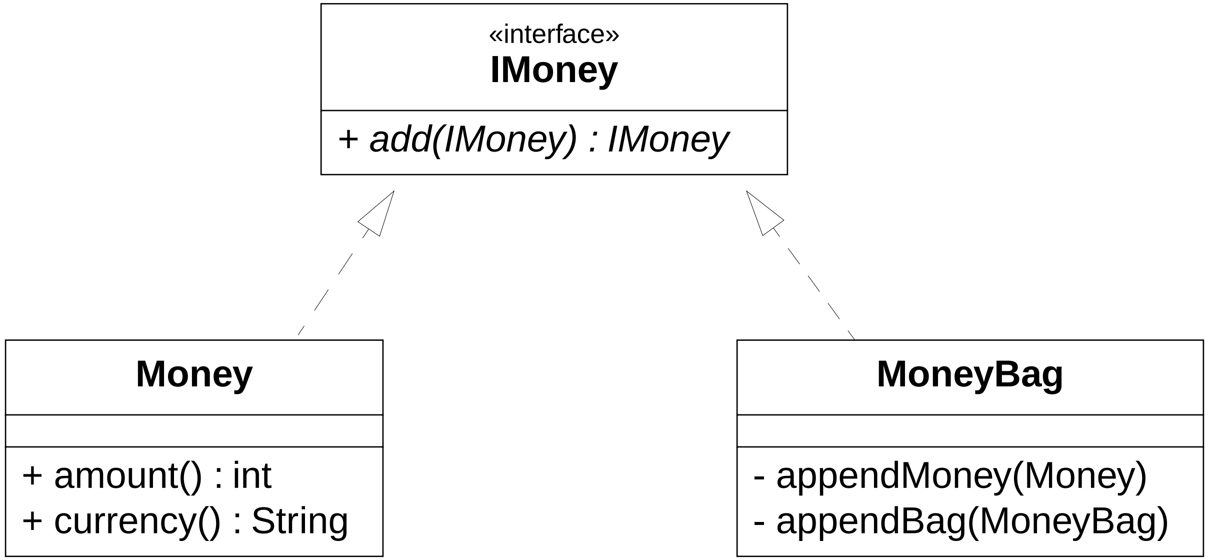
We would like to freely *add together* arbitrary *Monies and MoneyBags*, and be sure that *equals behave as equals:*

```
public void testMixedSimpleAdd() {
  // [12 CHF] + [7 USD] == {[12 CHF][7 USD]}
  Money bag[] = { f12CHF, f7USD };
  MoneyBag expected = new MoneyBag(bag);
  assertEquals(expected, f12CHF.add(f7USD));
}
```

That implies that Money and MoneyBag should implement a common interface ...

The IMoney interface (I)

Monies know how to be added to other Monies



Do we need anything else in the IMoney interface?

Double Dispatch (I)

How do we implement add() without breaking encapsulation?

> *“The idea behind double dispatch is to use an additional call to discover the kind of argument we are dealing with...”*

```
class Money implements IMoney { ...
  public IMoney add(IMoney m) {
    return m.addMoney(this);     // add me as a Money
  } ...
}

class MoneyBag implements IMoney { ...
  public IMoney add(IMoney m) {
    return m.addMoneyBag(this); // add me as a MoneyBag
  } ...
}
```

Double Dispatch (I)

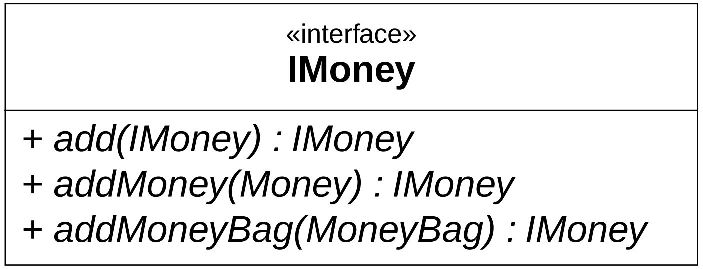
The rest is then straightforward ...

```
class Money implements IMoney { ...
  public IMoney addMoney(Money m) {
    if (m.currency().equals(currency()) )
      return new Money(amount()+m.amount(), currency());
    return new MoneyBag(this, m);
  }

  public IMoney addMoneyBag(MoneyBag s) {
    return s.addMoney(this);
  } ...
```

and MoneyBag takes care of the rest.

The IMoney interface (II)

So, the common interface is:



«interface»
IMoney

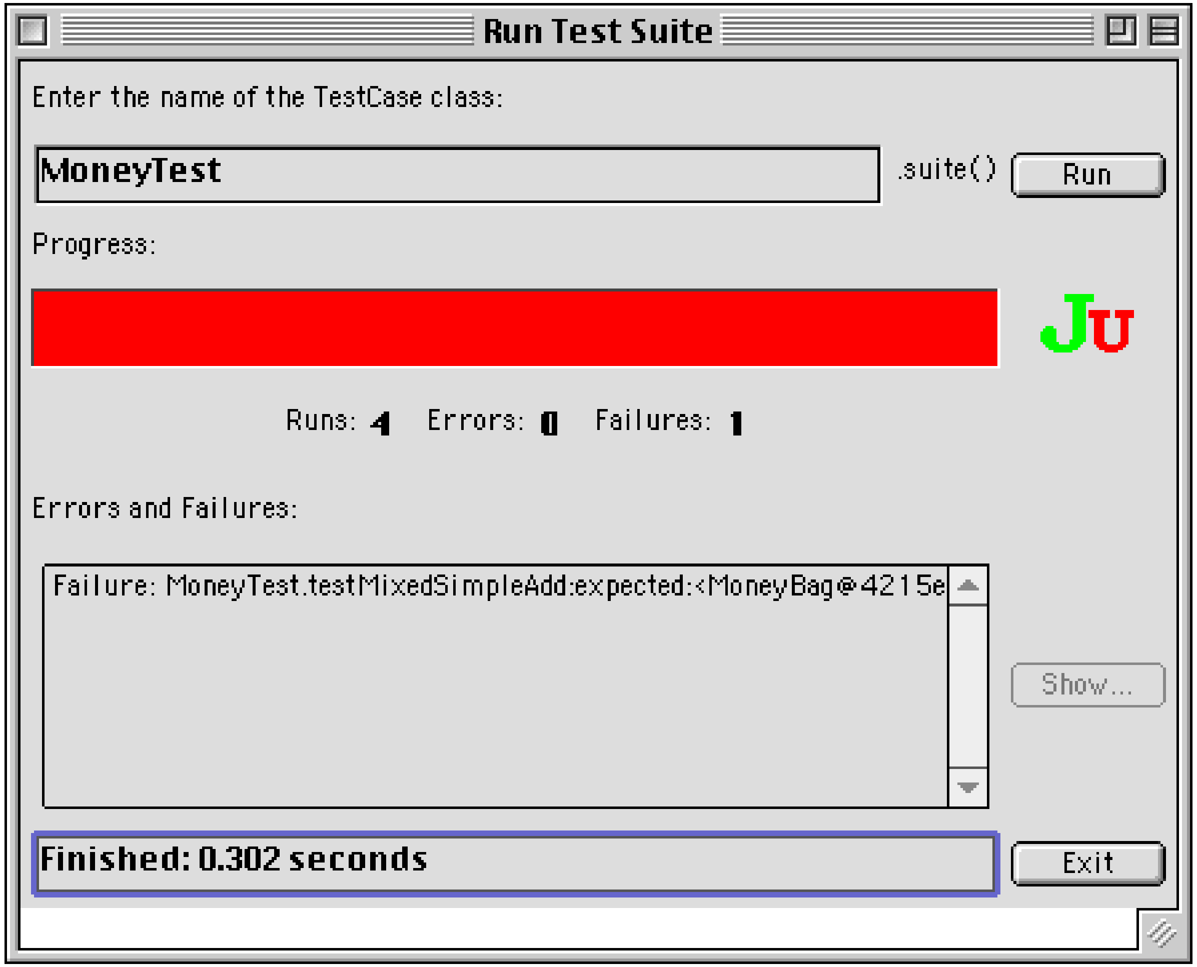
+ *add(IMoney) : IMoney*
+ *addMoney(Money) : IMoney*
+ *addMoneyBag(MoneyBag) : IMoney*

```
public interface IMoney {
  public IMoney add(IMoney aMoney);

  IMoney addMoney(Money aMoney);
  IMoney addMoneyBag(MoneyBag aMoneyBag);
}
```

NB: addMoney() and addMoneyBag() are only needed within the Money package.

A Failed test

This time we are not so lucky ...



Enter the name of the TestCase class:

MoneyTest .suite()

Progress:

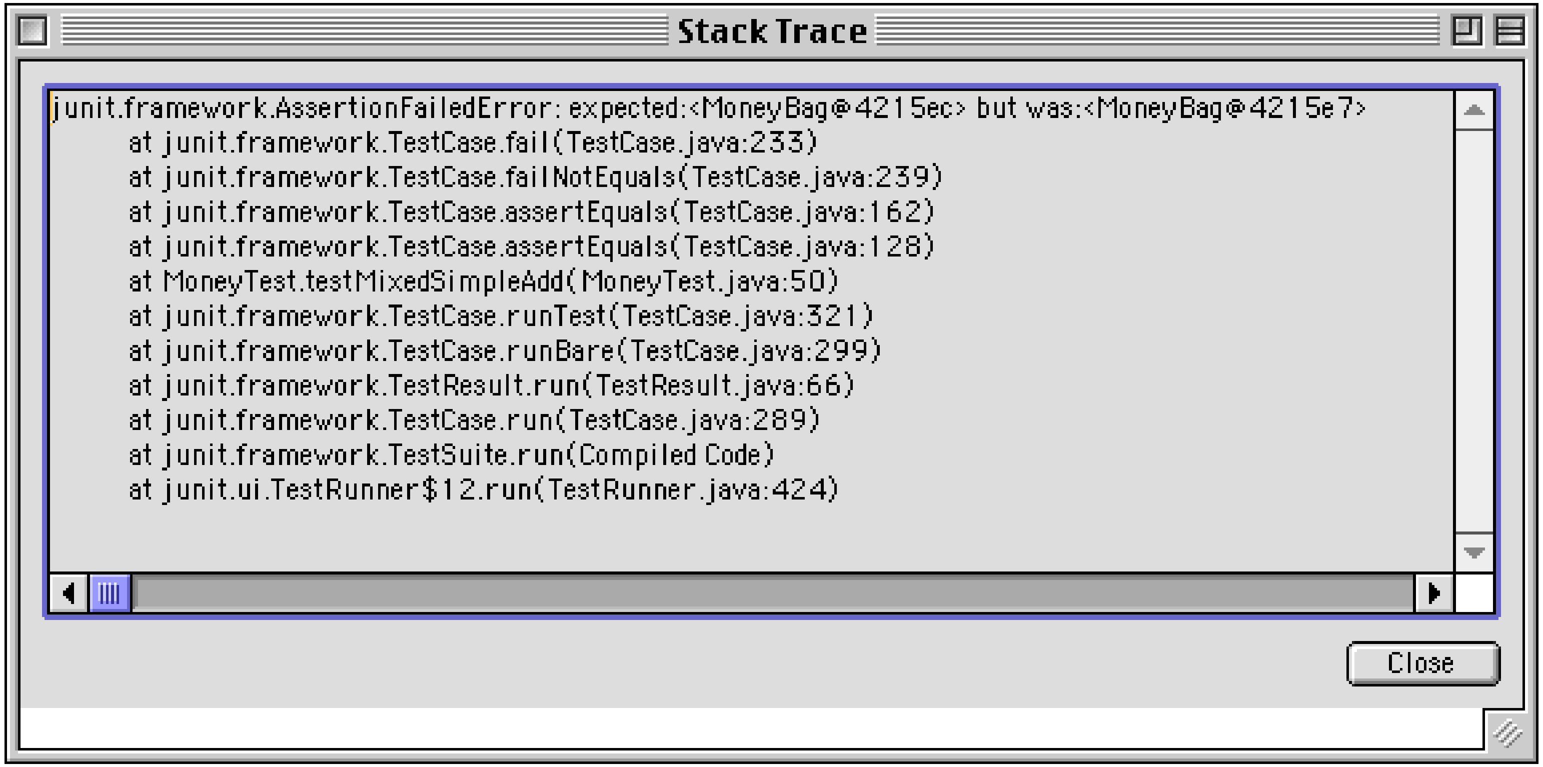
Runs: 4 Errors: 0 Failures: 1

Errors and Failures:

Failure: MoneyTest.testMixedSimpleAdd:expected:<MoneyBag@4215e

Finished: 0.302 seconds

Diagnostics



```
junit.framework.AssertionFailedError: expected:<MoneyBag@4215ec> but was:<MoneyBag@4215e7>
	at junit.framework.TestCase.fail(TestCase.java:233)
	at junit.framework.TestCase.failNotEquals(TestCase.java:239)
	at junit.framework.TestCase.assertEquals(TestCase.java:162)
	at junit.framework.TestCase.assertEquals(TestCase.java:128)
	at MoneyTest.testMixedSimpleAdd(MoneyTest.java:50)
	at junit.framework.TestCase.runTest(TestCase.java:321)
	at junit.framework.TestCase.runBare(TestCase.java:299)
	at junit.framework.TestResult.run(TestResult.java:66)
	at junit.framework.TestCase.run(TestCase.java:289)
	at junit.framework.TestSuite.run(Compiled Code)
	at junit.ui.TestRunner$12.run(TestRunner.java:424)
```

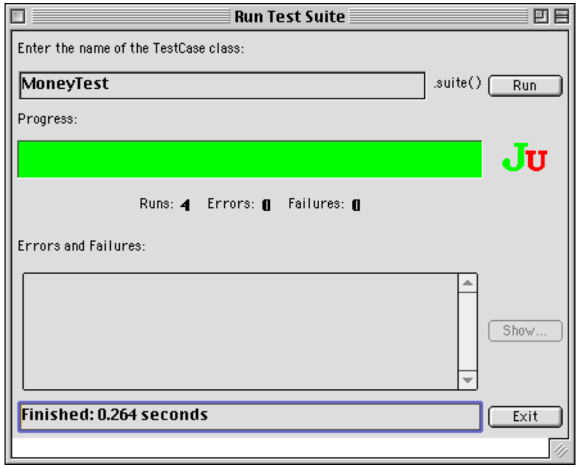
We quickly discover that we forgot to implement MoneyBag.equals()!

✎ *Why was this a run-time failure rather than a compile-time error?*

The fix ...

We fix it ...

```
class MoneyBag implements IMoney { ...
   public boolean equals(Object anObject) {
      if (anObject instanceof MoneyBag) {
         ...
      } else {
         return false;
      }
   }
```


Run Test Suite

Enter the name of the TestCase class: MoneyTest .suite()

Progress:

Runs: 4 Errors: 0 Failures: 0

Errors and Failures:

Finished: 0.264 seconds

... test it, and continue developing.

Summary

You should know the answers to these questions:

- ❑ How does a framework differ from a library?
- ❑ Why do TestCase and TestSuite implement the same interface?
- ❑ What is a unit test?
- ❑ What is a test “fixture”?
- ❑ What should you test in a TestCase?
- ❑ What is “double dispatch”? What does the name mean?

Can you answer the following questions?

- ✎ *How does the MoneyTest suite know which test methods to run?*
- ✎ *How does the TestRunner invoke the right suite() method?*
- ✎ *Why doesn’t the Java compiler complain that MoneyBag.equals() is used without being declared?*

8. Software Components: Collections

Overview

- ❑ Example problem: The Jumble Puzzle
- ❑ The Java 2 collections framework
- ❑ Interfaces: Collections, Sets, Lists and Maps
- ❑ Implementations ...
- ❑ Algorithms: sorting ...
- ❑ Iterators

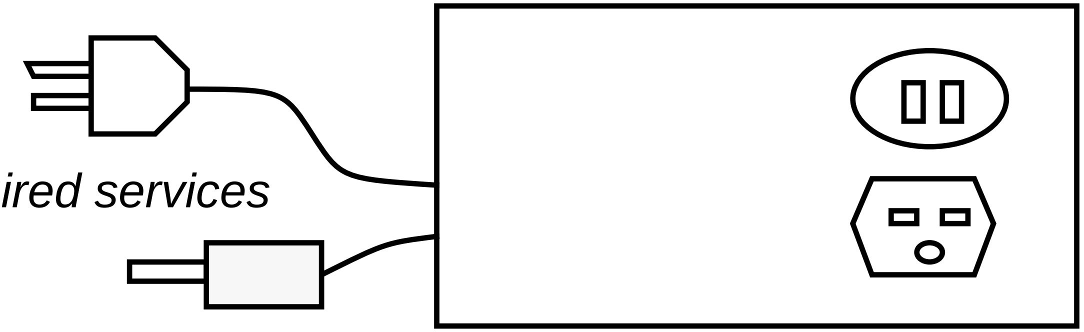
Source

- ❑ “Collections 1.2”, by Joshua Bloch, in *The Java Tutorial* , java.sun.com

Components

Components are *black-box* entities that:

- ❑ import *required* services and
- ❑ export *provided* services
- ❑ must be *designed to be composed*



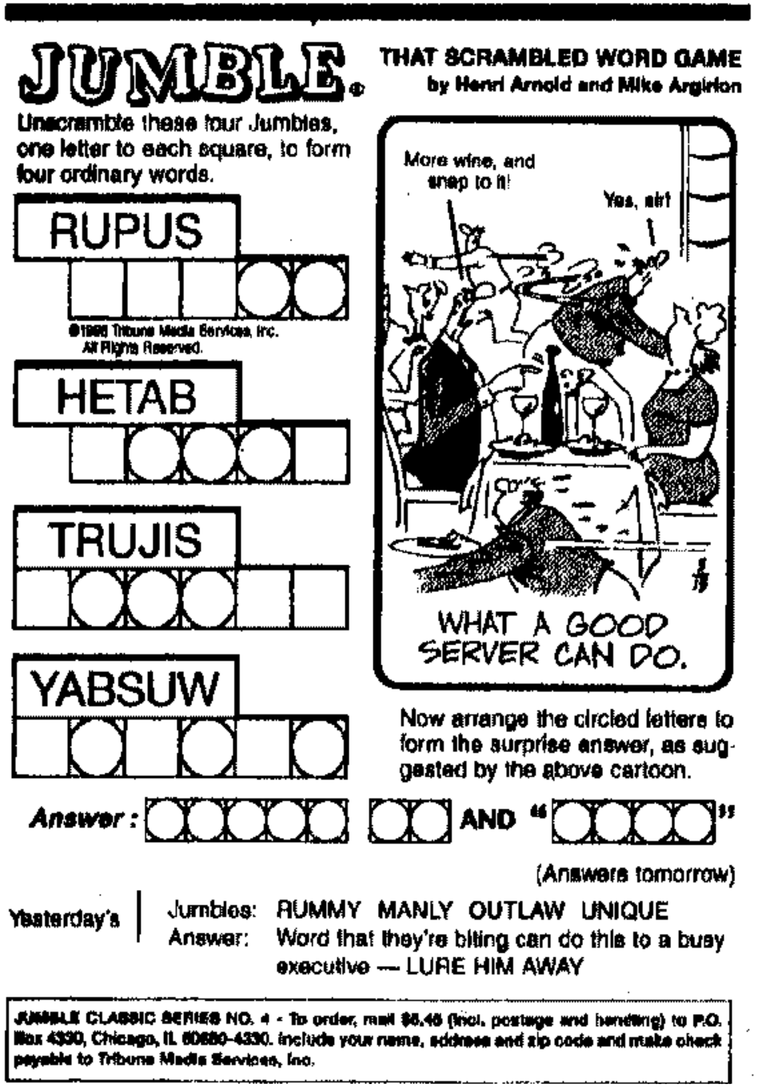
Components may be fine-grained (classes) or coarse-grained (applications).

The Jumble Puzzle

The Jumble Puzzle tests your English vocabulary by presenting four jumbled, ordinary words.

The circled letters of the unjumbled words represent the jumbled answer to a cartoon puzzle.

Since the jumbled words can be found in an electronic dictionary, it should be possible to write a program to automatically solve the first part of the puzzle (unjumbling the four words).



JUMBLE® THAT SCRAMBLED WORD GAME
by Henri Arnold and Mike Argirion

Unscramble these four Jumbles, one letter to each square, to form four ordinary words.

RUPUS
□□□○○

©1996 Tribune Media Services, Inc.
All Rights Reserved.

HETAB
□○○○□

TRUJIS
□○○○□□

YABSUW
□○□○□○

WHAT A GOOD SERVER CAN DO.

Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.

Answer: ○○○○○ ○○ AND "○○○○"

(Answers tomorrow)

Yesterday's | Jumbles: RUMMY MANLY OUTLAW UNIQUE
Answer: Word that they're biting can do this to a busy executive — LURE HIM AWAY

JUMBLE CLASSIC SERIES NO. 4 - To order, mail $8.45 (incl. postage and handling) to P.O. Box 4330, Chicago, IL 60680-4330. Include your name, address and zip code and make check payable to Tribune Media Services, Inc.

Naive Solution

Generate all permutations of the jumbled words:

rupus
urpus
uprus
purus
pruus
rpuus
ruups
urups
...

For each permutation, check if it exists in the word list:

abacus
abalone
abase
abash
...
zounds
zucchini
Zurich
zygote

The obvious, naive solution is extremely inefficient: a word with *n* characters may have up to n! permutations. A five-letter word may have 120 permutations and a six-letter word may have 720 permutations. “rupus” has 60 permutations.

✎ *Exactly how many permutations will a given word have?*

Rethinking the Jumble Problem

Observation: if a jumbled word (e.g. “rupus”) can be unjumbled to a real word in the list, then these two words are *jumbles of each other* (i.e. they are anagrams).

☞ Is there a fast way to tell if two words are anagrams?

Two words are anagrams if they are made up of the same set of characters.

☞ Each word has a unique “key” consisting of its letters in sorted order

The key for “rupus” is “prsuu”.

☞ Two words are anagrams if they have the same key

We can unjumble “rupus” by looking for a word with the same key.

An Efficient Solution

1. Build an associative array of keys and words for every word in the dictionary:

2. Generate the key of a jumbled word:
   key(“rupus”) = “prsuu”

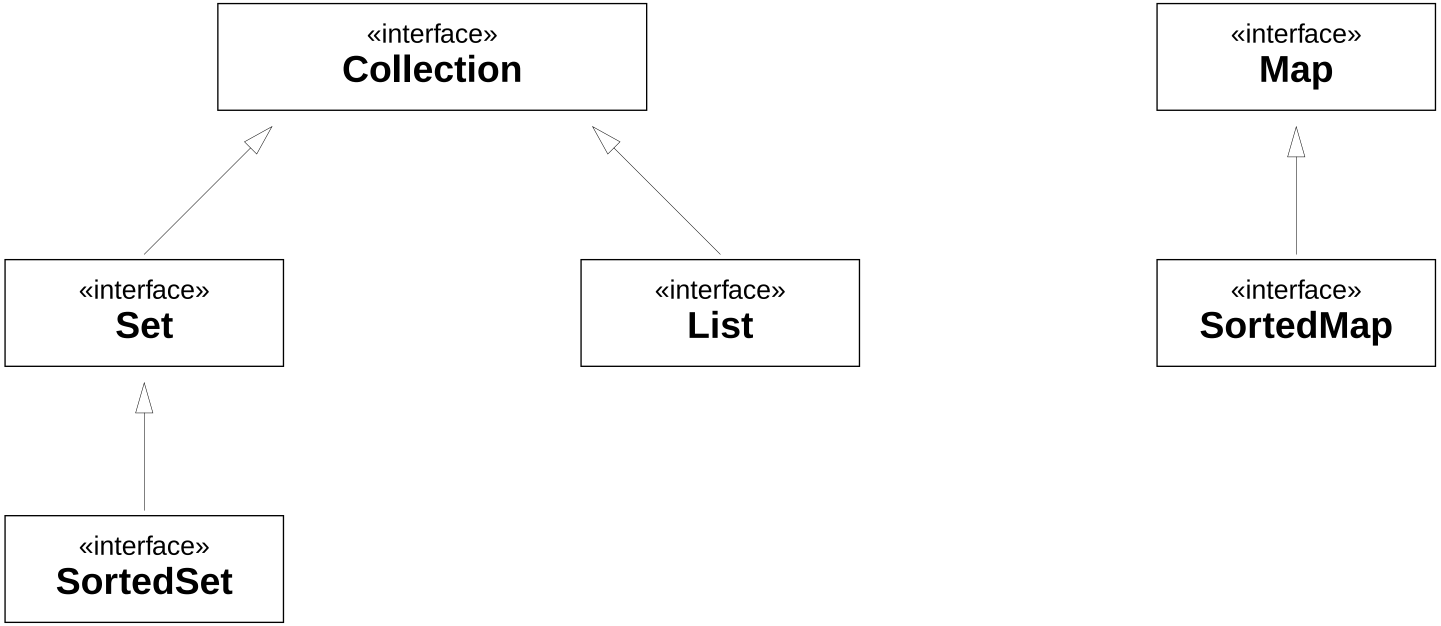
3. Look up and return the words with the same key.

Key	*Word*
aabcsu	abacus
aabelno	abalone
aabes	abase
aabhs	abash
...	...
dnosuz	zounds
cchiinuz	zucchini
chiruz	zurich
egotyz	zygote

To implement a software solution, we need associative arrays, lists, sort routines, and possibly other components.

The Collections Framework

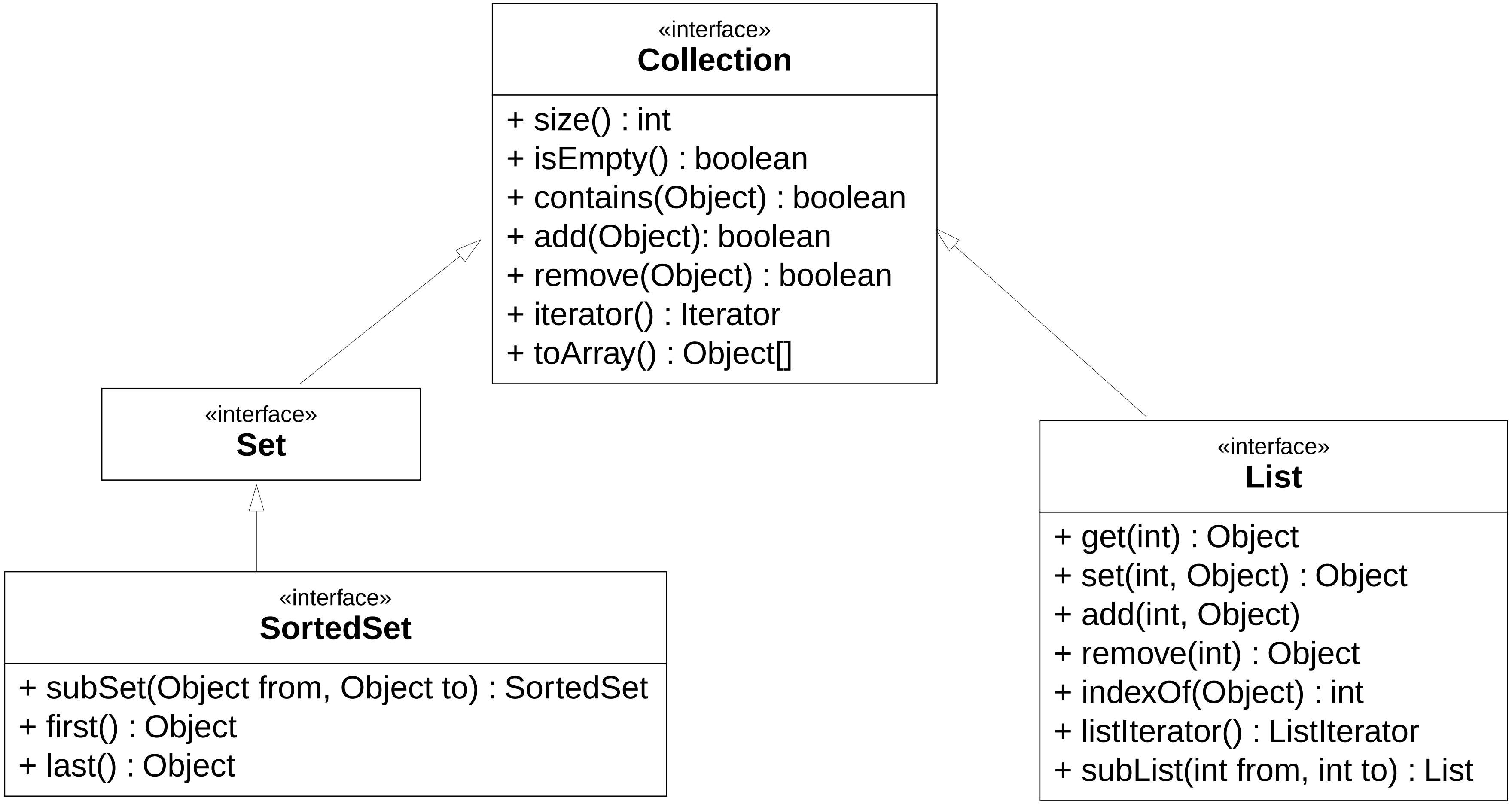
The Java Collections framework contains interfaces, implementations and algorithms for manipulating collections of elements.



Sets and Lists are kinds of collections.

Maps manage mappings from keys to values

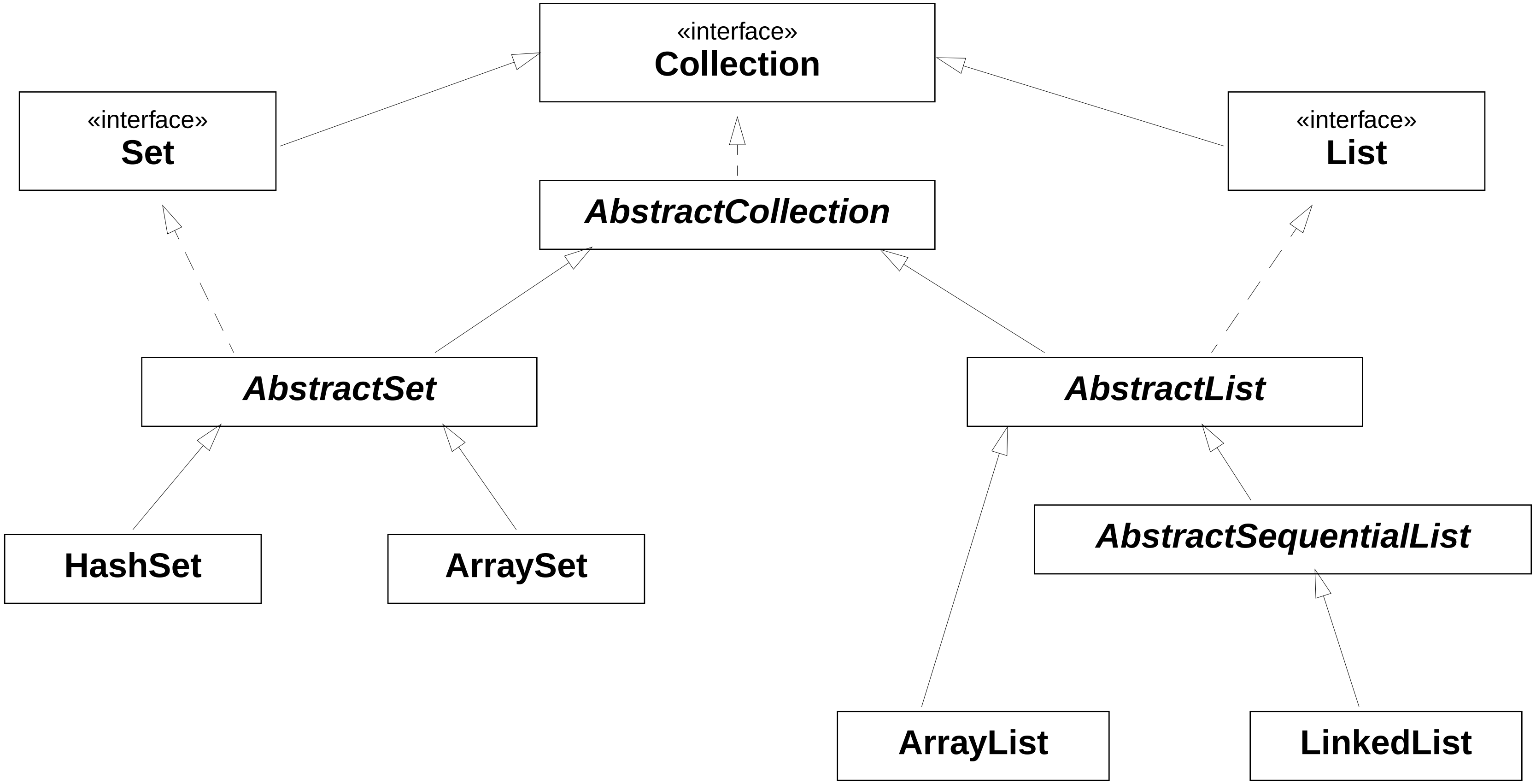
Collection Interfaces



Lists may contains duplicated elements. Sets may not.

Implementations

The framework provides at least two implementations of each interface.



✎ *Can you guess how the standard implementations work?*

Interfaces and Abstract Classes

Principles at play:

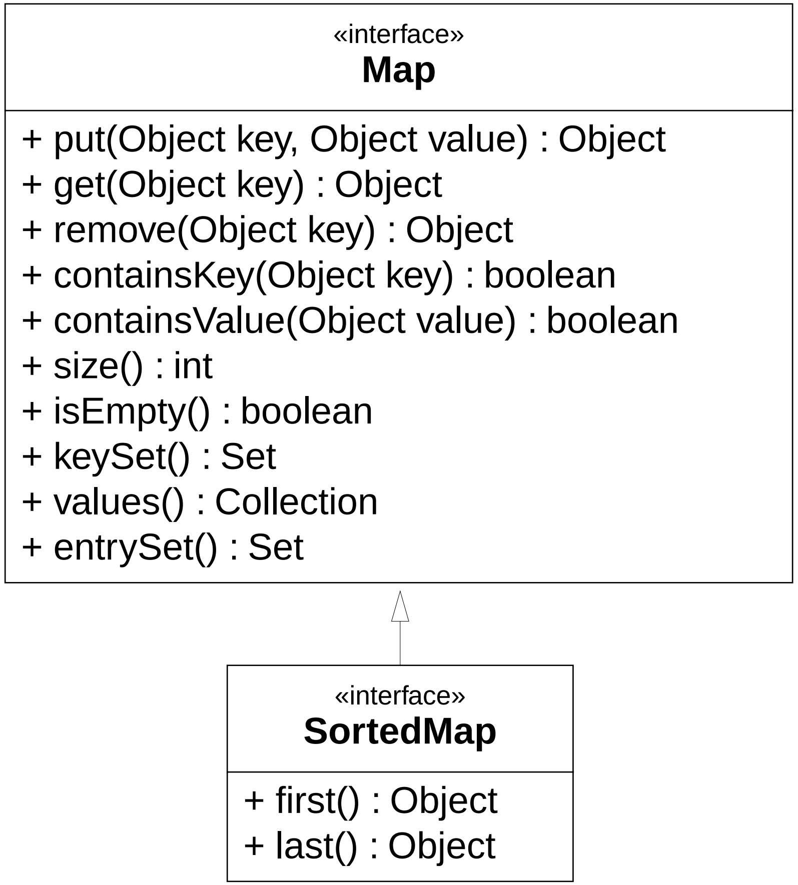
- ❑ Clients *depend only on interfaces*, not classes
- ❑ Classes may *implement multiple interfaces*
- ❑ Single inheritance doesn’t prohibit *multiple subtyping*
- ❑ Abstract classes collect *common behaviour* shared by multiple subclasses
  - ☞ but cannot be instantiated themselves (makes no sense)

Maps

A *Map* is an object that manages a set of (key, value) pairs.

A *Sorted Map* maintains its entries in ascending order.

Map is implemented by HashMap and TreeMap.



«interface»
Map

+ put(Object key, Object value) : Object
+ get(Object key) : Object
+ remove(Object key) : Object
+ containsKey(Object key) : boolean
+ containsValue(Object value) : boolean
+ size() : int
+ isEmpty() : boolean
+ keySet() : Set
+ values() : Collection
+ entrySet() : Set

«interface»
SortedMap

+ first() : Object
+ last() : Object

Jumble

We can implement the Jumble dictionary as a kind of HashMap:

```
public class Jumble extends HashMap {
  public static void main(String args[]) {
    if (args.length == 0) {
      System.err.println("Usage: java Jumble <wordfile>");
      return;
    }
    Jumble wordMap = null;
    try {
      wordMap = new Jumble(args[0]);
    } catch (IOException err) {
      System.err.println("Can't load dictionary " + args[0]);
      return;
    }
    wordMap.inputLoop();
  } ...
}
```

Jumble constructor

A Jumble dictionary knows the file containing the words to load ...

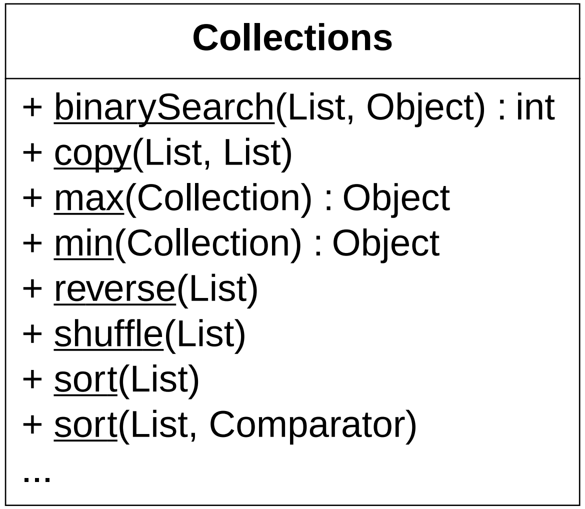
```
private String _wordFile;

Jumble(String wordFile) throws IOException {
  super();
  _wordFile = wordFile;
  loadDictionary();
}
```

Before we continue, we need a way to generate a key for each word ...

Algorithms

The Collections framework provides various algorithms, such as sorting and searching, that *work uniformly for all kinds of Collections and Lists*.

(Also any that you define yourself!)



Collections

+ binarySearch(List, Object) : int
+ copy(List, List)
+ max(Collection) : Object
+ min(Collection) : Object
+ reverse(List)
+ shuffle(List)
+ sort(List)
+ sort(List, Comparator)
...

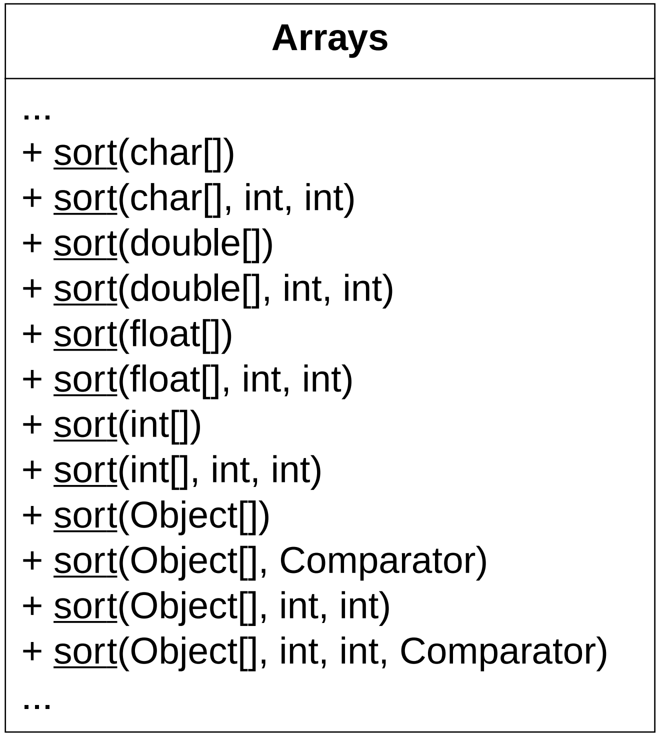
These algorithms are *static methods* of the Collections class.

✎ *As a general rule, static methods should be avoided in an OO design. Are there any good reasons here to break this rule?*

Array algorithms

There is also a class, Arrays, consisting of static methods for searching and sorting that operate on Java arrays of basic data types.

✎ *Which sort routine should we use to generate unique keys for the Jumble puzzle?*



Arrays

...
+ sort(char[])
+ sort(char[], int, int)
+ sort(double[])
+ sort(double[], int, int)
+ sort(float[])
+ sort(float[], int, int)
+ sort(int[])
+ sort(int[], int, int)
+ sort(Object[])
+ sort(Object[], Comparator)
+ sort(Object[], int, int)
+ sort(Object[], int, int, Comparator)
...

Sorting characters

The easiest solution is to convert the word to an array of characters, sort that, and convert the result back to a String.

```
public static String sortKey(String word) {
  char [] letters = word.toCharArray();
  Arrays.sort(letters);
  return new String(letters);
}
```

✎ *What other possibilities do we have?*

Loading the dictionary

Reading the dictionary is straightforward ...

```
    private void loadDictionary() throws IOException {
      BufferedReader in =
        new BufferedReader(new FileReader(_wordFile));
      String word = in.readLine();
      while (word != null) {
        this.addPair(sortKey(word), word);
        word = in.readLine();
      }
    }
```

... but there may be a *List* of words for any given key!

```
    private void addPair(String key, String word) {
      List wordList = (List) this.get(key);
      if (wordList == null)
        wordList = new ArrayList();
      wordList.add(word);
      this.put(key, wordList);
    }
```

The input loop

does the obvious ...

```
public void inputLoop() { ...
    System.out.print("Enter a word to unjumble: ");
    String word;
    while ((word = in.readLine()) != null) {
      ...
        List wordList = (List) this.get(sortKey(word));
        if (wordList == null) {
          System.out.println("Can't unjumble " + word);
        } else {
          System.out.println(
              word + " unjumbles to: " + wordList);
        } ...
      System.out.print("next word: ");
    } ...
}
```

Running the unjumbler ...

```
Enter a word to unjumble: rupus
rupus unjumbles to: [usurp]
Enter a word to unjumble: hetab
hetab unjumbles to: [bathe]
next word: please
please unjumbles to: [asleep, elapse, please]
next word: java
Can't unjumble java
next word:
Quit? (y/n): y
bye!
```

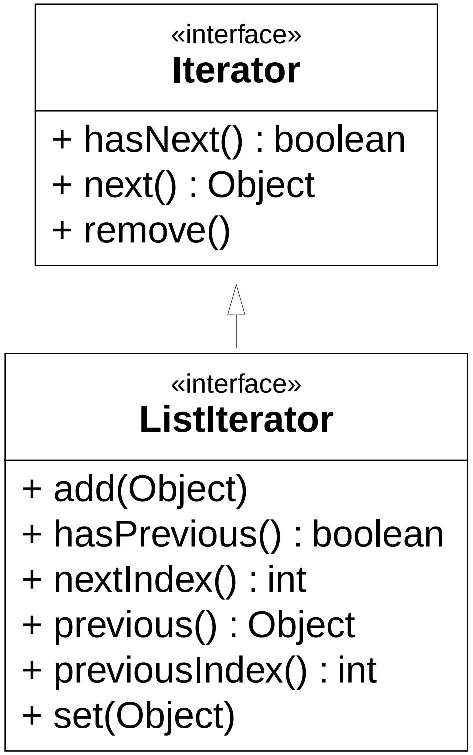
Iterators

➤ How do you iterate through a Collection whose elements are unordered?

✔ *Use an iterator.*

An *Iterator* is an object that lets you walk through an arbitrary collection, whether it is ordered or not.

Lists additionally provide *ListIterators* that allows you to traverse the list in either direction and modify the list during iteration.



Iterating through the key set

We can use iterators to find the key with the largest set of associated anagrams:

```
public List maxAnagrams() {
   int max = 0;
   List anagrams = null;
   Iterator keys = this.keySet().iterator();
   while (keys.hasNext()) {
      String key = (String) keys.next();
      List words = (List) this.get(key);
      if (words.size() > max) {
         anagrams = words;
         max = words.size();
      }
   }
   return anagrams;
}
```

Printing wordMap.maxAnagrams() yields: [caret, carte, cater, crate, trace]

How to use the framework

- ❑ If you need collections in your application, *stick to the standard interfaces*.
- ❑ Use one of the *default implementations*, if possible
- ❑ If you need a specialized implementation, make sure it is *compatible* with the standard ones, so you can mix and match
- ❑ Make your applications depend only on the collections *interfaces*, if possible, not the concrete classes
- ❑ Always use the *least specific* interface that does the job (Collection, if possible)

Summary

You should know the answers to these questions:

- ❑ How are Sets and Lists similar? How do they differ?
- ❑ Why is Collection an interface rather than a class?
- ❑ Why are the sorting and searching algorithms implemented as *static* methods?
- ❑ What is an iterator? What problem does it solve?

Can you answer the following questions?

- ✎ *Of what use are the AbstractCollection, AbstractSet and AbstractList?*
- ✎ *Why doesn’t Map extend Collection?*
- ✎ *Why does the Jumble constructor call super()?*
- ✎ *Which implementation of Map will make Jumble run faster? Why?*

9. GUI Construction

Overview

- ❑ Applets
- ❑ Model-View-Controller
- ❑ AWT Components, Containers and Layout Managers
- ❑ Events and Listeners
- ❑ Observers and Observables

Sources

- ❑ David Flanagan, *Java in a Nutshell*, O'Reilly, 1996
- ❑ Mary Campione and Kathy Walrath, *The Java Tutorial* , The Java Series, Addison-Wesley, 1996

A Graphical TicTacToe?

Our existing TicTacToe implementation is very limited:

- ❑ single-user at a time
- ❑ textual input and display

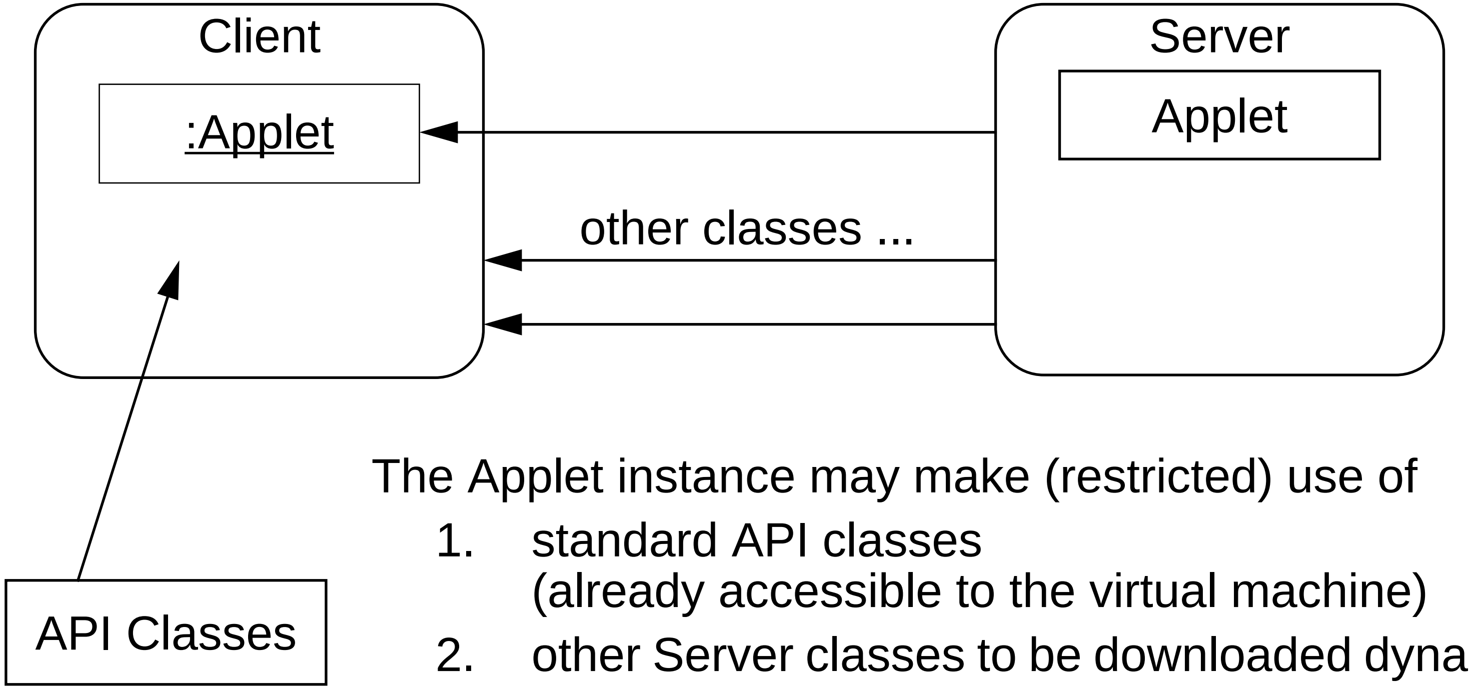
We would like to migrate it towards an interactive, network based game:

- ❑ players on separate machines
- ❑ running the game as an “applet” in a browser
- ❑ with graphical display and mouse input

As first step, we will migrate the game to run as an applet

Applets

Applet *classes* can be downloaded from an HTTP server and instantiated by a client. When instantiated, the Applet will be `initialized` and `started` by the client.



The Applet instance may make (restricted) use of

1. standard API classes
   (already accessible to the virtual machine)
2. other Server classes to be downloaded dynamically.

java.applet.Applet extends java.awt.Panel and can be used to construct a UI ...

The Hello World Applet

The simplest Applet:

```
import java.awt.*;                          // for Graphics
import java.applet.Applet;
public class HelloApplet extends Applet {
  public void init() { repaint(); }         // request a refresh
  public void paint( Graphics g ) {
    g.drawString( "Hello World!", 30, 30 );
  }
} // NB: there is no main() method!
```

HTML applet inclusion:

```
<title>TrivialApplet</title>
<hr>
<applet archive="AppletClasses.jar"
code="HelloApplet.class" width=200 height=200>
</applet>
<hr>
```

Accessing the game as an Applet

The compiled TicTacToe classes will be made available in a directory “AppletClasses” on our web server.

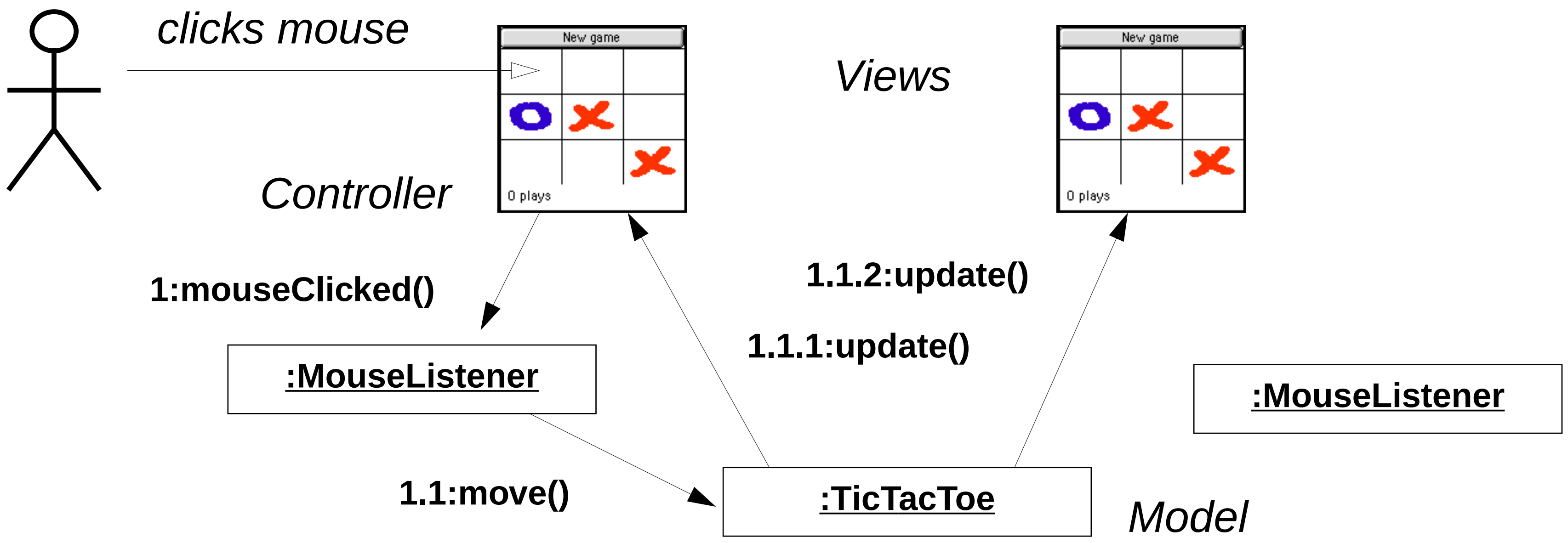
```
<title>GameApplet</title>
<hr>
<applet
  codebase="AppletClasses"
  code="tictactoe.GameApplet.class"
  width=200
  height=200>
</applet>
<hr>
```

`GameApplet` extends `java.applet.Applet`.

Its `init()` will instantiate and connect the other game classes ...

Model-View-Controller

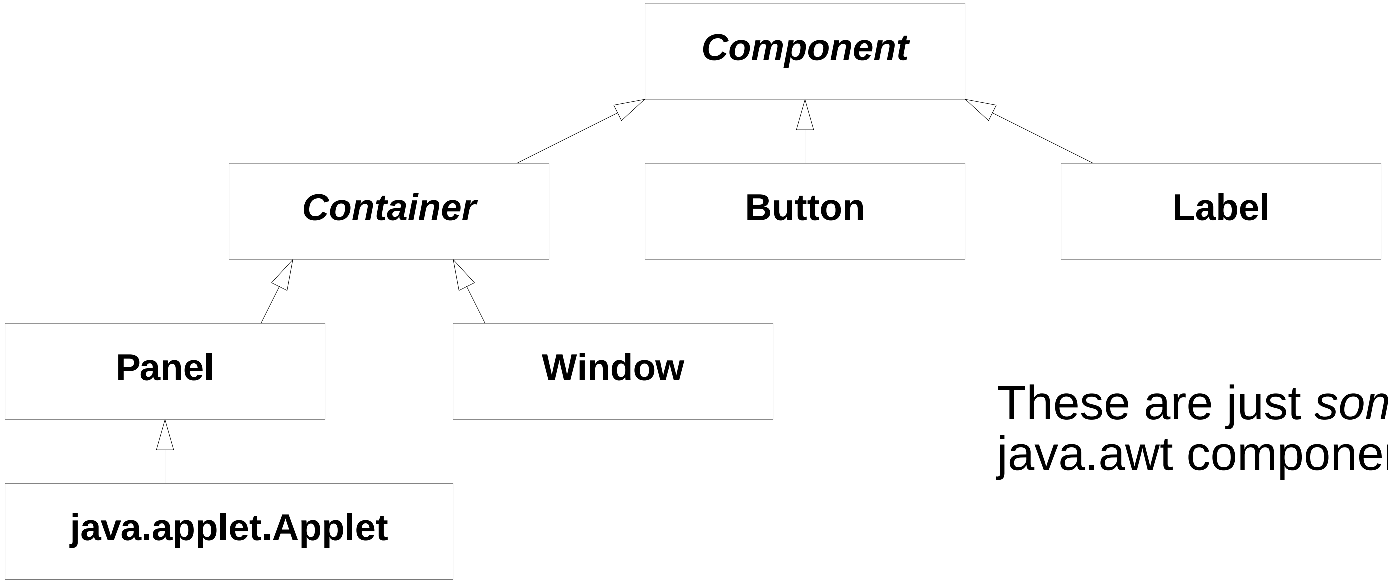
Version 1.6 of our game implements a *model* of the game, without a GUI.
The GameApplet will implement a graphical *view* and a *controller* for GUI events.



The MVC paradigm separates an application from its GUI so that multiple views can be dynamically connected and updated.

AWT Components and Containers

The java.awt package defines GUI *components*, *containers* and their *layout managers.*



These are just *some* of the java.awt components ...

A *Container* is a component that may contain other components.

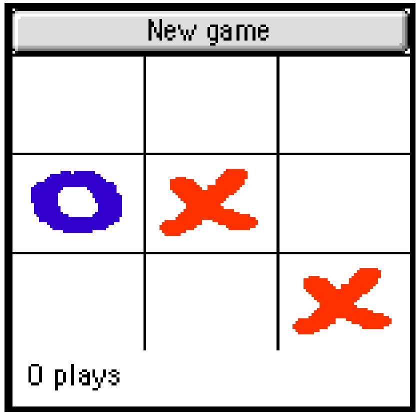
A Panel is a container inside another container. (E.g., an Applet inside a browser.)

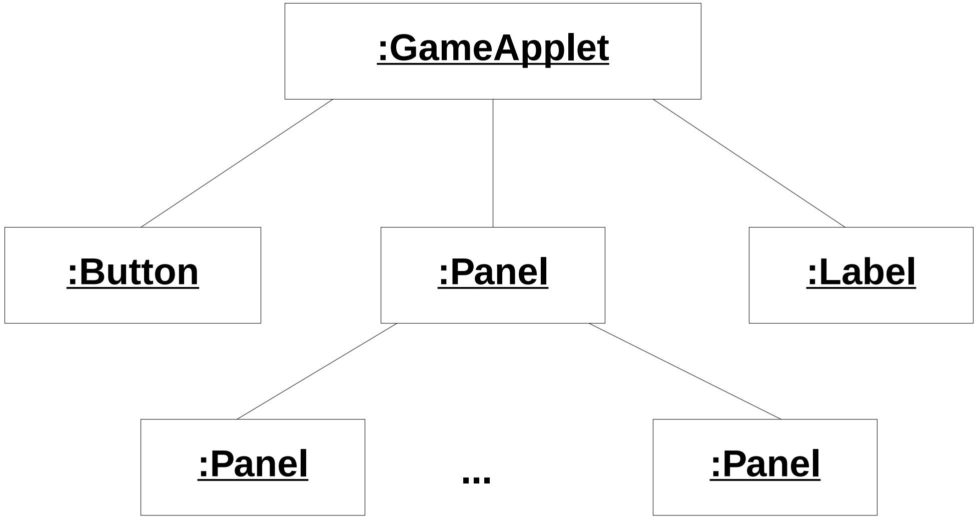
A Window is a top-level container.

NB: There are also many graphics classes to define colours, fonts, images etc.

The GameApplet

The GameApplet is a Panel using a BorderLayout (with a centre and up to four border components), and containing a Button (“North”), a Panel (“Center”) and a Label (“South”).





The central Panel itself contains a grid of squares (Panels) and uses a GridLayout.

Other layout managers are FlowLayout, CardLayout and GridBagLayout ...

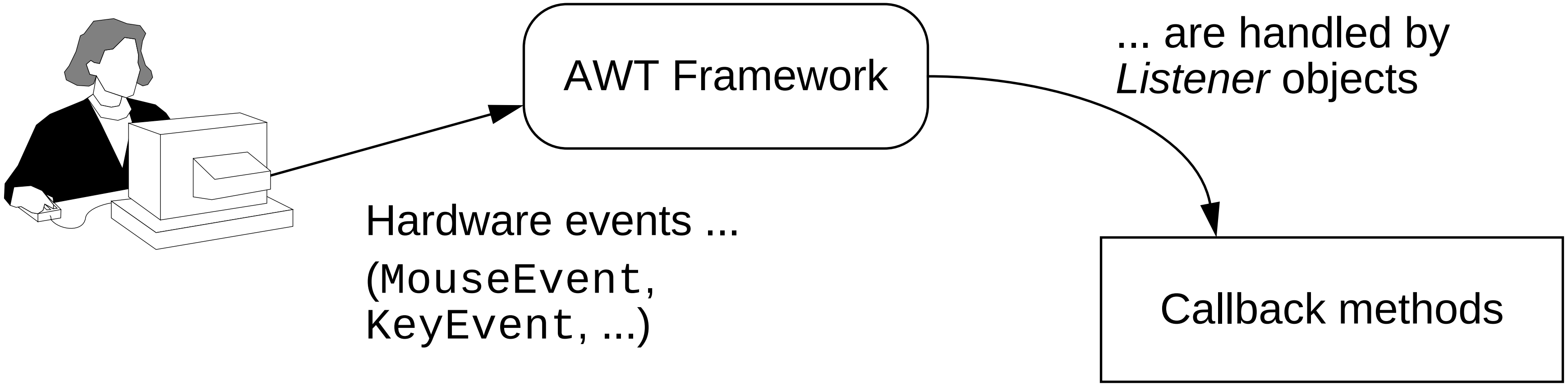
Laying out the GameApplet

Instantiate the game, initialize the view, and connect the view to the model ...

```
public void init() {
  _game = ...
  setLayout(new BorderLayout());
  setSize(MINSIZE*_game.cols(),MINSIZE*_game.rows());
  add("North", makeControls());
  add("Center", makeGrid());
  _label = new Label();
  add("South", _label);
  showFeedBack(_game.currentPlayer().mark() + " plays");
}

private Component makeControls() {
  Button again = new Button("New game");
  ...
  return again;
}
```

Events and Listeners (I)

Instead of actively checking for GUI events, you can define callback methods that will be invoked when your GUI objects receive events:



AWT Framework

... are handled by *Listener* objects

Hardware events ...

(`MouseEvent`, `KeyEvent`, ...)

Callback methods

AWT Components *publish* events and Listeners *subscribe* interest in them.

Events and Listeners (II)

Every AWT component publishes a variety of different events (defined in java.awt.event).

Component	*Events*	*Listener Interface*	*Listener methods*
Button	ActionEvent	ActionListener	actionPerformed()

Component	MouseEvent	MouseListener	mouseClicked()
			mouseEntered()
			mouseExited()
			mousePressed()
			mouseReleased()

		MouseMotionListener	mouseDragged()
			mouseMoved()

	KeyEvent	KeyListener	keyPressed()
			keyReleased()
			keyTyped()

...			

Each event class has its associated listener interfaces.

Listening for Button events

When we create the “New game” Button, we attach an ActionListener with the Button.addActionListener() method:

```
private Component makeControls() {
  Button again = new Button("New game");
  again.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      showFeedBack("starting new game ...");
      newGame(); // clear the board and bind to a new game
    }
  });
  return again;
}
```

Instead of creating a separate, named subclass of ActionListener, we can instantiate a so-called *anonymous inner class,* which implements the required interface using methods of the enclosing class.

Listening for mouse clicks

We must similarly attach a MouseListener to each Place on the board.

```
private Component makeGrid()
{ ...
  Panel grid = new Panel();
  grid.setLayout(new GridLayout(rows, cols)); ...
  for (int row=rows-1; row>=0; row--) {
    for (int col=0; col<cols; col++) {
      Place p = new Place(col, row, xImage, oImage);
      p.addMouseListener(new PlaceListener(p, this));
      p.setBackground(Color.white);
      grid.add(p);
      _places[col][row] = p;
    }
  }
  return grid;
}
```

NB: we could have multiple Listeners subscribe to the same events ...

The PlaceListener

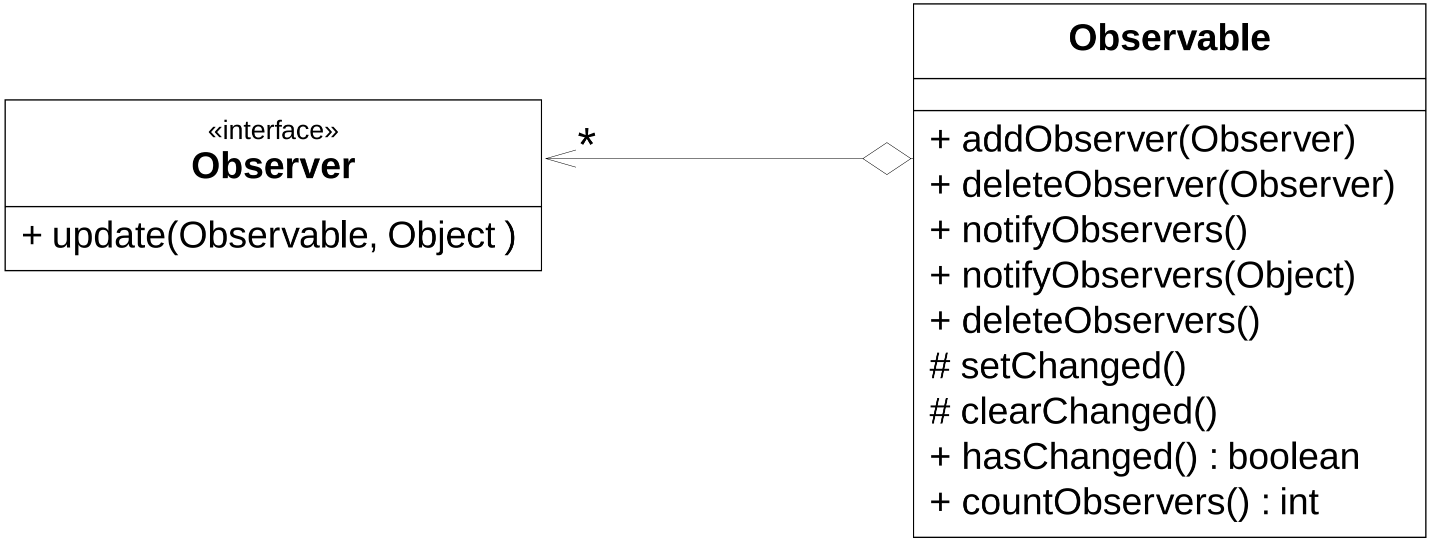
MouseAdapter is a convenience class that defines *empty* MouseListener methods.

We only have to define the mouseClicked() method:

```
public class PlaceListener extends MouseAdapter { ...
  public void mouseClicked(MouseEvent e) { ...
    if (game.notOver()) {
      try {
        ((AppletPlayer) game.currentPlayer()).move(col,row);
      } catch (AssertionException err) {
        _applet.showFeedBack("Invalid move ignored (" ... );
      }
      if (!game.notOver()) {
        _applet.showFeedBack("Game over -- "
          + game.winner() + " wins!");
      }
    } else { _applet.showFeedBack("The game is over!"); }
  }
}
```

Observers and Observables

A class can implement the java.util.Observer interface when it wants to be informed of changes in Observable objects.



An Observable object can have one or more Observers.

After an observable instance changes, calling notifyObservers() causes all observers to be notified by means of their update() method.

Observing the BoardGame

```
public class GameApplet extends Applet implements Observer
{ ...
  public void update(Observable o, Object arg) {
    Move move = (Move) arg;
    showFeedBack("got an update: " + move);
    _places[move.col][move.row].setMove(move.player);
  }
}
public abstract class AbstractBoardGame
    extends Observable implements BoardGame
{ ...
  public void move(int col, int row, Player p)
    throws AssertionException
  { ...
    setChanged();
    notifyObservers(new Move(col, row, p));
  }
}
```

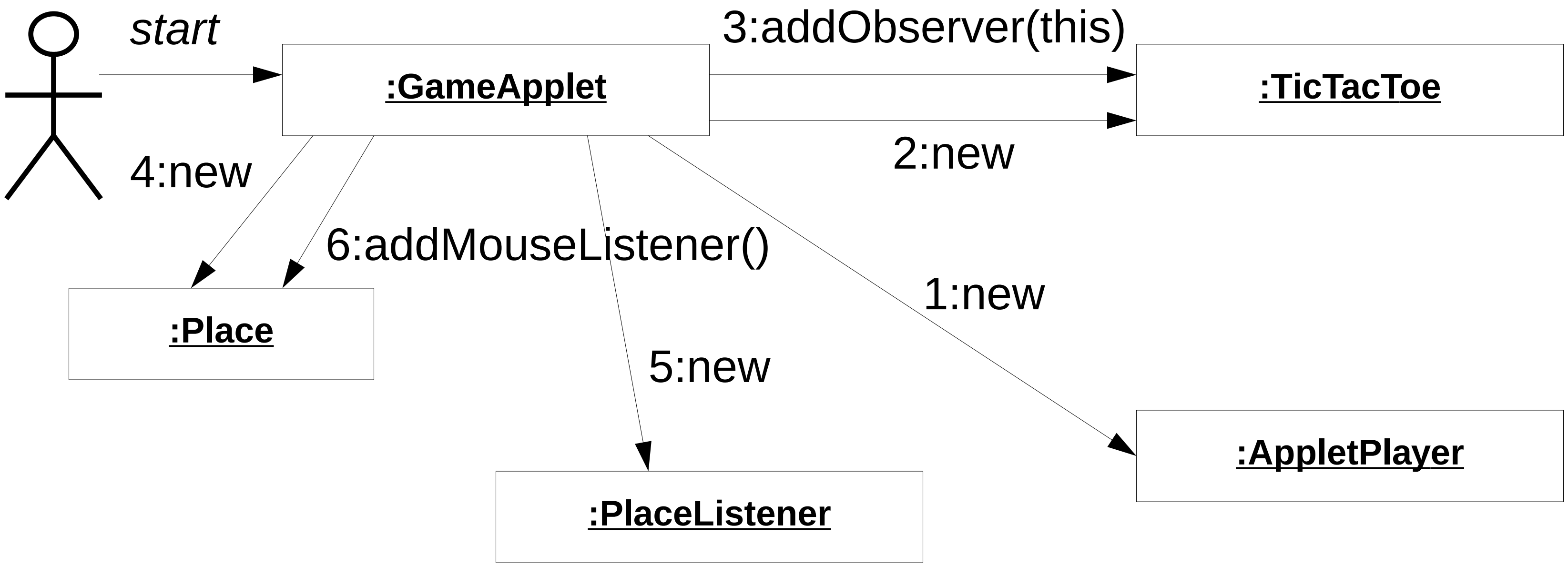
Communicating changes

A Move instance bundles together information about a change of state in a BoardGame. Sent by a BoardGame to its Observers:

```
public class Move {
  public final int col;
  public final int row;
  public final Player player;
  public Move(int col, int row, Player player) {
    this.col = col;
    this.row = row;
    this.player = player;
  }
  public String toString() {
    return "Move(" + col + "," + row + "," + player + ")";
  }
}
```

Setting up the connections

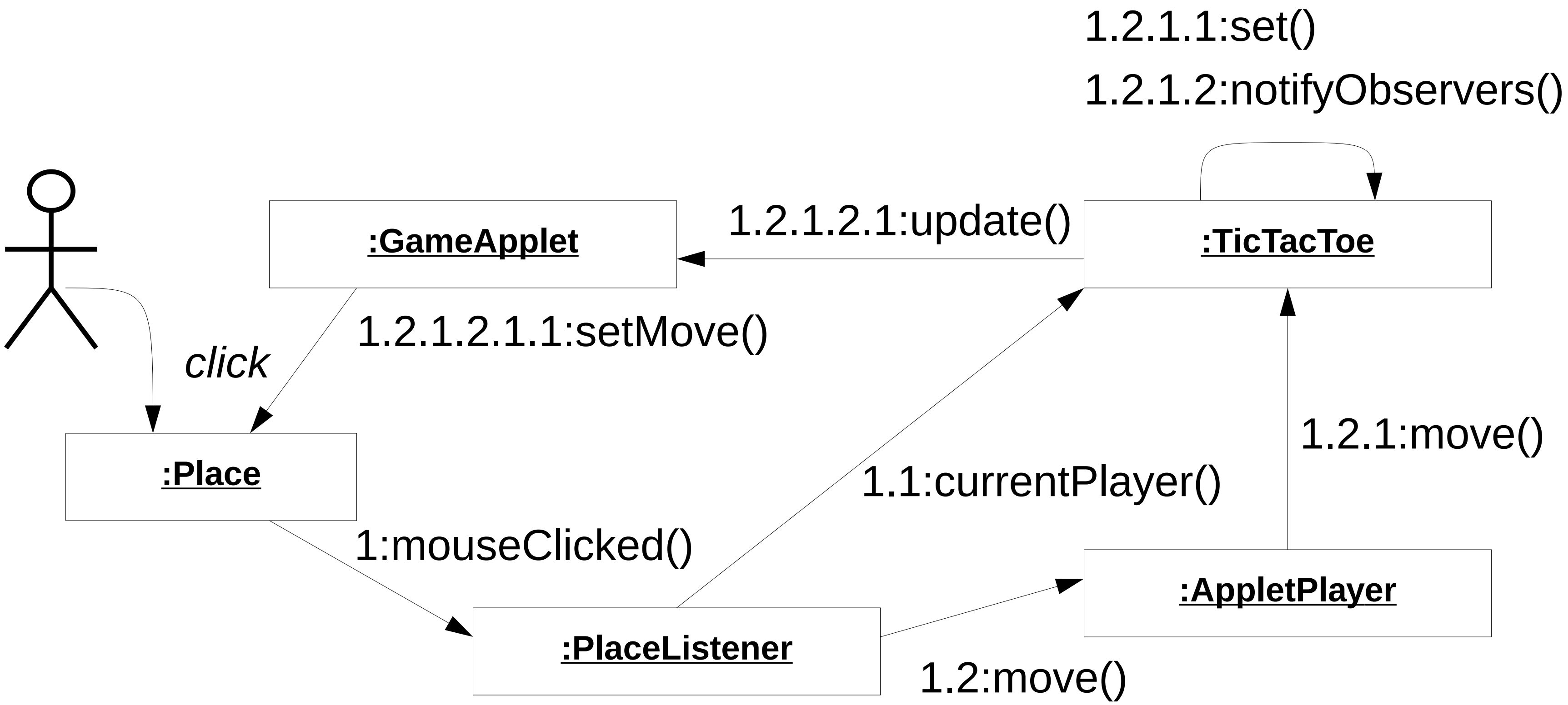
When the GameApplet is loaded, its init() method is called, causing the model, view and controller components to be instantiated.



The GameApplet subscribes itself as an Observer to the game, and subscribes a PlaceListener to MouseEvents for each Place on the view of the BoardGame.

Playing the game

Mouse clicks are propagated from a Place (controller) to the BoardGame (model):



If the corresponding move is valid, the model's state changes, and the GameApplet updates the Place (view).

Refactoring the BoardGame

Adding a GUI to the game affects many classes. We iteratively introduce changes, and rerun our tests after every change ...

- ❑ Shift responsibilities between BoardGame and Player (both should be passive!)
  - ☞ introduce Player interface, InactivePlayer and StreamPlayer classes
  - ☞ move getRow() and getCol() from BoardGame to Player
  - ☞ move BoardGame.update() to GameDriver.playGame()
  - ☞ change BoardGame to hold a matrix of Players, not marks
- ❑ Introduce Applet classes (GameApplet, Place, PlaceListener)
  - ☞ Introduce AppletPlayer
  - ☞ PlaceListener triggers AppletPlayer to move
- ❑ BoardGame must be observable
  - ☞ Introduce Move to communicate changes from BoardGame to Observer

GUI objects in practice ...

Use Swing, not AWT

- ❑ javax.swing provides a set of “lightweight” (all-Java language) components that (more or less!) work the same on all platforms.

Use a GUI builder

- ❑ Interactively build your GUI rather than programming it — add the hooks later.

Summary

You should know the answers to these questions:

- ❑ Why doesn't an Applet need a `main()` method?
- ❑ What are models, view and controllers?
- ❑ Why does Container extend Component and not vice versa?
- ❑ What does a layout manager do?
- ❑ What are events and listeners? Who publishes and who subscribes to events?
- ❑ The TicTacToe game knows nothing about the GameApplet or Places. How is this achieved? Why is this a good thing?

Can you answer the following questions?

- ✎ *How could you get Applets to download objects instead of just classes?*
- ✎ *How could you make the game start up in a new Window?*
- ✎ *What is the difference between an event listener and an observer?*
- ✎ *The Move class has public instance variables — isn't this a bad idea?*
- ✎ *What kind of tests would you write for the GUI code?*

10. Clients and Servers

Overview

- ❑ RMI — Remote Method Invocation
- ❑ Remote interfaces
- ❑ Serializable objects
- ❑ Synchronization
- ❑ Threads
- ❑ Compiling and running an RMI application

Sources

- ❑ David Flanagan, *Java Examples in a Nutshell*, O'Reilly, 1997
- ❑ "RMI 1.2", by Ann Wollrath and Jim Waldo, in *The Java Tutorial* , java.sun.com

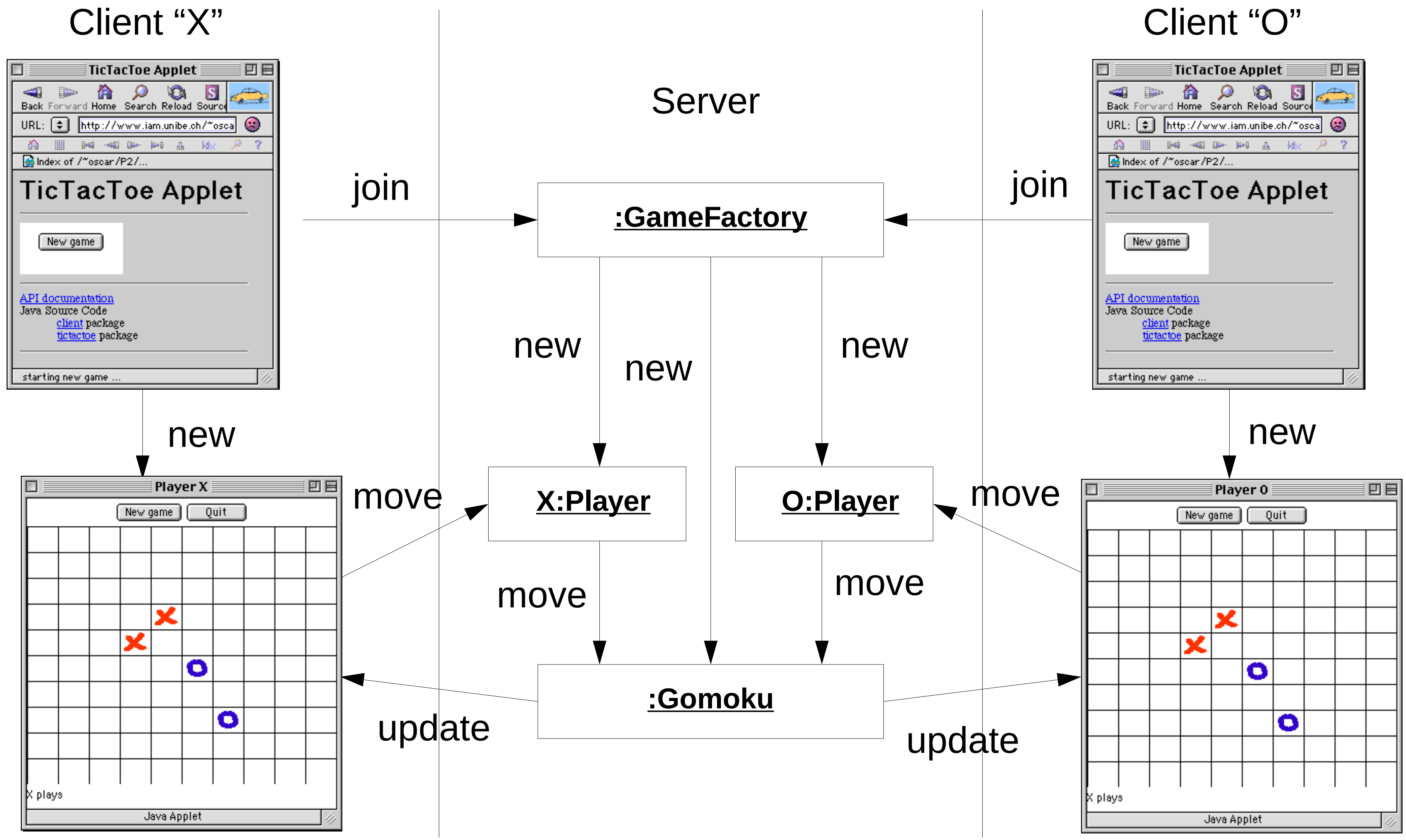
A Networked TicTacToe?

We now have a usable GUI for our game, but it still supports only a single user.

We would like to support:

- ❑ players on separate machines
- ❑ each running the game as an applet in a browser
- ❑ with a “game server” managing the state of the game

The concept



Client “X”

Server

Client “O”

:GameFactory

X:Player

O:Player

:Gomoku

TicTacToe Applet

API documentation
Java Source Code
client package
tictactoe package

starting new game ...

X plays

The problem

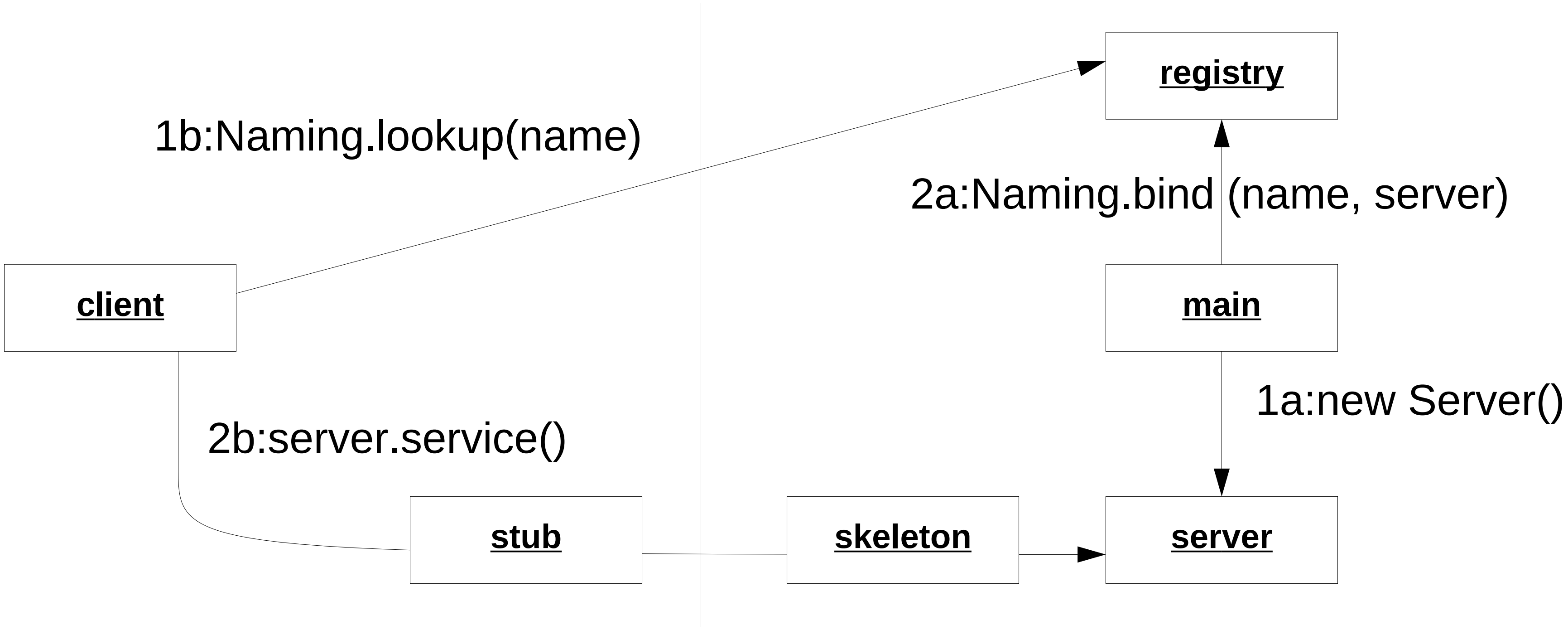
Unfortunately Applets alone are not enough to implement this scenario!

We must answer several questions:

- ❑ Who creates the GameFactory?
- ❑ How does the Applet connect to the GameFactory?
- ❑ How do the server objects connect to the client objects?
- ❑ How do we download objects (rather than just classes)?
- ❑ How do the server objects synchronize concurrent requests?

Remote Method Invocation

RMI allows an application to *register* a Java object under a public *name* with an RMI *registry* on the server machine.



A client may *look up* up the service using the public name, and obtain a local object that acts as a proxy for the remote server object.

Remote method invocations are managed by a local *stub* and a remote *skeleton*.

Developing an RMI application

There are several steps to using RMI:

1. Implement a server
   - ☞ Decide which objects will be remote servers and *specify their interfaces*
   - ☞ Implement the server objects
2. Implement a client
   - ☞ Clients must use the remote interfaces
   - ☞ Objects passed as parameters must be *serializable*
3. Compile and install the software
   - ☞ Use the rmic compiler to *generate stubs and skeletons* for remote objects
4. Run the application
   - ☞ Start the RMI registry
   - ☞ Start and register the servers
   - ☞ Start the client

Designing client/server interfaces

Interfaces between clients and servers should be as small as possible.

Low coupling:

- ❑ simplifies development and debugging
- ❑ maximizes independence
- ❑ reduces communication overhead

We split the game into three packages:

- ❑ **client** — contains the GUI components, the EventListeners and the Observer
- ❑ **server** — contains the *server interfaces* and the communication classes
- ❑ **tictactoe** — contains the model and the *server implementation* classes

NB: The client’s Observer must be updated from the server side, so is also a “server”!

Identifying remote interfaces

To implement the distributed game, we need three interfaces:

RemoteGameFactory

- ❑ called by the client to *join a game*
- ❑ implemented by tictactoe.GameFactory

RemoteGame

- ❑ called by the client to *query the game state* and to *handle moves*
- ❑ implemented by tictactoe.Gameproxy
  - ☞ we simplify the game interface by hiding Player instances

RemoteObserver

- ❑ called by the server to *propagate updates*
- ❑ implemented by client.GameObserver

Specifying remote interfaces

To define a remote interface:

- ❑ the interface must *extend* java.rmi.Remote
- ❑ every method must be declared to *throw* java.rmi.RemoteException
- ❑ every argument and return value must:
  - ☞ be a primitive data type (int, etc.), or
  - ☞ be declared to implement java.io.Serializable, or
  - ☞ implement a Remote interface

RemoteGameFactory

This is the interface used by clients to join a game.

If a game already exists, the client joins the existing game. Else a new game is made.

```
public interface RemoteGameFactory extends Remote {
  public RemoteGame joinGame() throws RemoteException;
}
```

The object returned implements the RemoteGame interface.

RMI will automatically create a stub on the client side and skeleton on the server side for the RemoteGame

RemoteGame

The RemoteGame interface hides all details of BoardGames and Players.
It exports only what is needed to implement the client:

```
public interface RemoteGame extends Remote {
  public boolean ready() throws RemoteException;
  public char join() throws RemoteException;
  public boolean move(Move move) throws RemoteException;
  public int cols() throws RemoteException;
  public int rows() throws RemoteException;
  public char currentPlayer() throws RemoteException;
  public String winner() throws RemoteException;
  public boolean notOver() throws RemoteException;
  public void addObserver(RemoteObserver o)
    throws RemoteException;
}
```

NB: To keep things simple, we avoid introducing a RemotePlayer interface.

RemoteObserver

This is the only interface the client exports to the server:

```
public interface RemoteObserver extends Remote {
  public void update(Move move) throws RemoteException;
}
```

NB: RemoteObserver is not compatible with java.util.Observer, since update() may throw a RemoteException ... We will have to bridge the incompatibility on the server side.

Serializable Objects

Objects to be passed as values must be declared to implement java.io.Serializable.

```
public class Move implements java.io.Serializable {
  public final int col;
  public final int row;
  public final char mark;
  public Move(int col, int row, char mark) {
    this.col = col;
    this.row = row;
    this.mark = mark;
  }
  public String toString() {
    return "Move(" + col + "," + row + "," + mark + ")";
  }
}
```

Move encapsulates the minimum information to communicate between client and server.

Implementing Remote objects

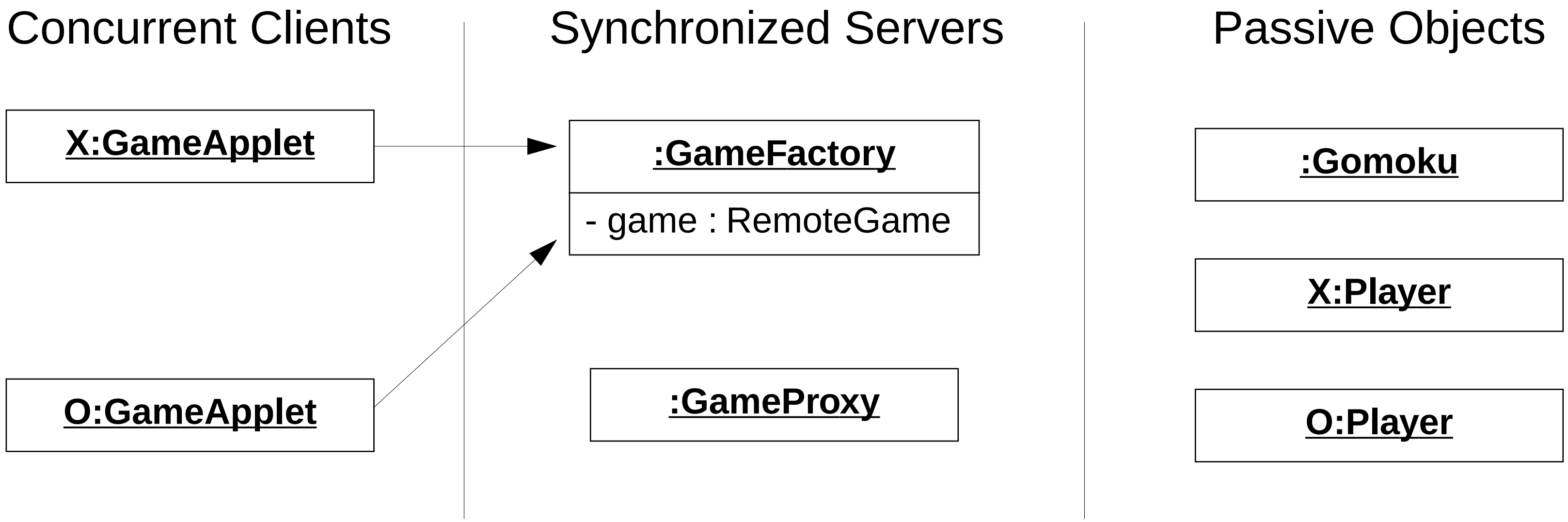
Remote objects should extend java.rmi.server.UnicastRemoteObject:

```
public class GameFactory extends UnicastRemoteObject
  implements RemoteGameFactory
{ private RemoteGame _game;
  public static void main(String[] args) { ... }
  public GameFactory() throws RemoteException { super(); }
  public synchronized RemoteGame joinGame()
    throws RemoteException
  { RemoteGame game = _game;
    if (game == null) { // first player => return new game
      game =  new GameProxy( new Gomoku( ...));
      _game = game;
    } else { _game = null; } // second player => join game
    return game;
  }
}
```

NB: All constructors for Remote objects must throws RemoteException!

A simple view of synchronization

A *synchronized* method obtains a lock for its object before executing its body.



➤ How can servers protect their state from concurrent requests?

✔ *Declare their public methods as synchronized.*

Make sure that synchronized objects don't call each other, or you may get a deadlock!

Registering a remote object

To bootstrap the server, we need a main() method that instantiates a GameFactory and registers it with a running RMI registry.

There must be a security manager installed so that RMI can safely download classes!

```
public static void main(String[] args) {
  if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
    System.out.println("Set new Security manager");
  }
  if (args.length != 1) { ... }
  String name = "//" + args[0] + "/GameFactory";
  try {
    RemoteGameFactory factory = new GameFactory();
    Naming.rebind(name, factory);
  } catch (Exception e) { ... }
}
```

The argument is the host id and port number of the registry (e.g., asterix.unibe.ch:2001)

GameProxy

The GameProxy interprets Moves and protects the client from any AssertionExceptions:

```
public class GameProxy extends UnicastRemoteObject
    implements RemoteGame
{ ...
  public synchronized boolean move(Move move)
    throws RemoteException
  { Player current = _game.currentPlayer();
    if (current.mark() != move.mark) return false;
    try {
      _game.move(move.col, move.row, current);
      return true; // the move succeeded
    } catch (AssertionException e) { return false; }
  } ...
}
```

Using Threads to protect the server

WrappedObserver adapts a RemoteObserver to implement java.util.Observer:

```
class WrappedObserver implements Observer {
  private RemoteObserver _remote;
  WrappedObserver(RemoteObserver ro) { _remote = ro; }
  public void update(Observable o, Object arg) {
    final Move move = (Move) arg; // final for inner class
    Thread doUpdate = new Thread() {
      public void run() {
        try {
          _remote.update(move);
        } catch(RemoteException err) { }
      }
    };
    doUpdate.start(); // start the Thread; ignore results
  }
}
```

The server must not block trying to update the client, so we use a new Thread!

Refactoring the BoardGame ...

Most of the changes were on the GUI side:

- ❑ defined separate client, server and tictactoe packages
- ❑ no changes to Drivers, Players, Runner, TicTactoe or Gomoku from 2.0
  - ☞ except renaming AppletPlayer to PassivePlayer (used only on server side)
- ❑ added BoardGame methods player() and addObserver()
  - ☞ added WrappedObserver to adapt RemoteObserver
- ❑ added remote interfaces and remote objects
- ❑ changed all client classes
  - ☞ separated GameApplet from GameView (to allow multiple views)
  - ☞ modified view to use Move and RemoteGame instead of Player

Compiling the code

We compile the source packages as usual, and install the results in a web-accessible location so that the GameApplet has access to the client and server .class files.

In addition, the client and the server need access to the *stub* and *skeleton* class files.

On Unix, chdir to the directory containing the client and tictactoe class file hierarchies

```
rmic -d . tictactoe.GameFactory
rmic -d . tictactoe.GameProxy
rmic -d . client.GameObserver
```

This will generate stub and skeleton class files for the remote objects.
(I.e., GameFactory_Skel.class etc.)

NB: Move is not a remote object, so we do not need to run rmic on its class file.

Running the application

We simply start the RMI registry on the host (asterix):

```
rmiregistry 2001 &
```

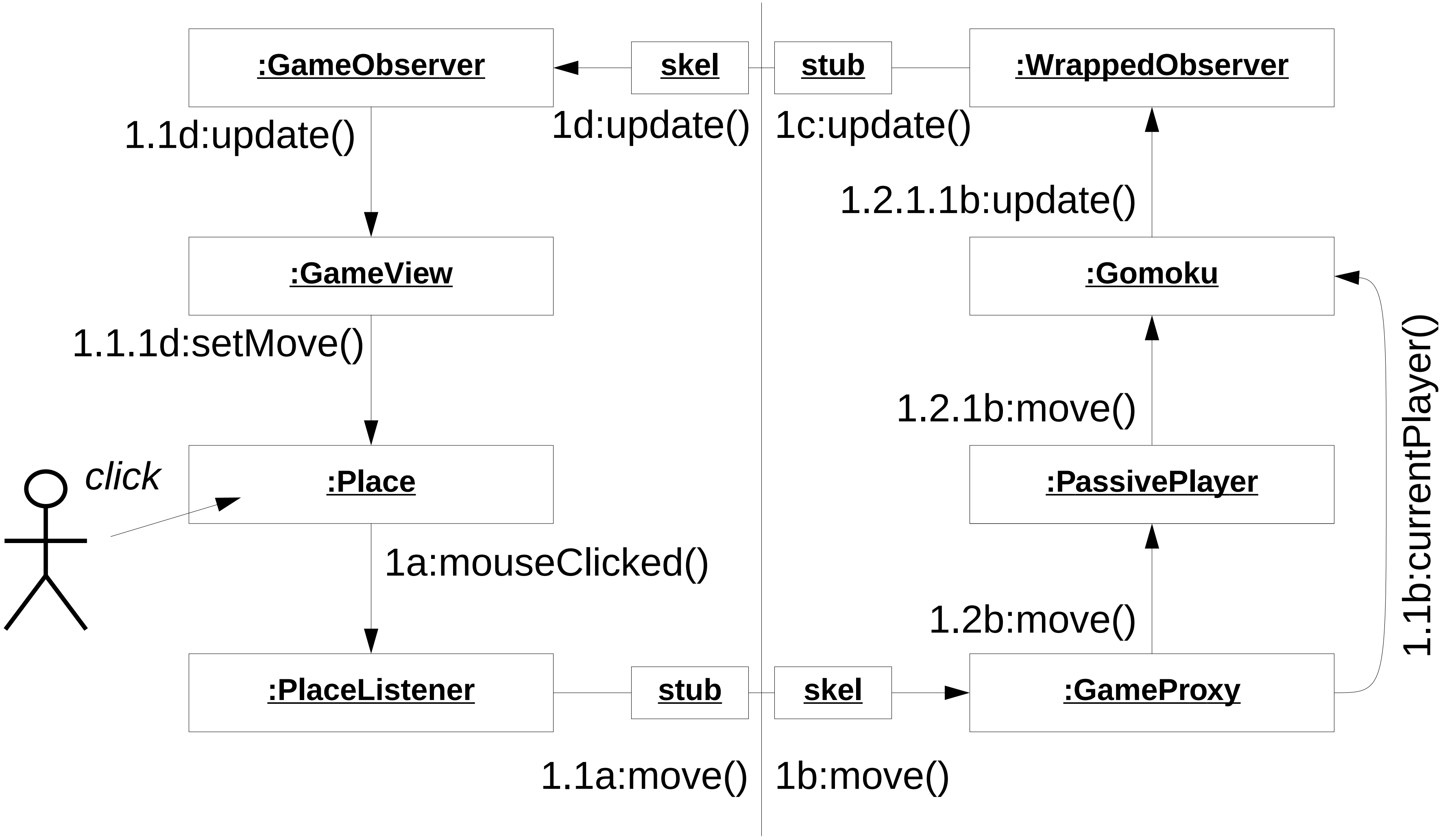
Start and register the servers:

```
setenv CLASSPATH ./classes
java -Djava.rmi.server.codebase=http://www.iam.unibe.ch/.../classes/ \
        tictactoe.GameFactory asterix.unibe.ch:2001
```

And start the clients with a browser or an appletviewer ...

NB: the RMI registry needs the codebase so it can instantiate the stubs and skeletons!

Playing the game



Other approaches

CORBA

- ❑ for non-java components

COM (DCOM, Active-X ...)

- ❑ for talking to MS applications

Sockets

- ❑ for talking other TCP/IP protocols

Software buses

- ❑ for sharing information across multiple applications

Summary

You should know the answers to these questions:

- ❑ How do you make a remote object available to clients?
- ❑ How does a client obtain access to a remote object?
- ❑ What are stubs and skeletons, and where do they come from?
- ❑ What requirements must a remote interface fulfil?
- ❑ What is the difference between a remote object and a serializable object?
- ❑ Why do servers often start new threads to handle requests?

Can you answer the following questions?

- ✎ *Suppose we modified the view to work with Players instead of Moves. Should Players then be remote objects or serializable objects?*
- ✎ *Why don’t we have to declare the AbstractBoardGame methods as synchronized?*
- ✎ *What kinds of tests would you write for the networked game?*
- ✎ *How would you extend the game to notify users when a second player is connected?*

11. Guidelines, Idioms and Patterns

Overview

- ❑ Programming style: Code Talks; Code Smells
- ❑ Idioms, Patterns and Frameworks
- ❑ Basic Idioms
  - ☞ Delegation, Super, Interface
- ❑ Basic Patterns
  - ☞ Adapter, Proxy, Template Method, Composite, Observer

Sources

- ❑ Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.
- ❑ Frank Buschmann, et al., *Pattern-Oriented Software Architecture — A System of Patterns*, Wiley, 1996
- ❑ Mark Grand, *Patterns in Java*, Volume 1, Wiley, 1998
- ❑ Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997
- ❑ “Code Smells”, http://c2.com/cgi/wiki?CodeSmells

Style

Code Talks

- ❑ Do the simplest thing you can think of (KISS)
  - ☞ Don't over-design
  - ☞ Implement things once and only once
  - ☞ First do it, then do it right, then do it fast
(don’t optimize too early)

- ❑ Make your intention clear
  - ☞ Write small methods
  - ☞ Each method should do one thing only
  - ☞ Name methods for what they do, not how they do it
  - ☞ Write to an interface, not an implementation

Refactoring

Redesign and refactor when the code starts to “smell”

Code Smells

- ❑ Methods too long or too complex
  - ☞ decompose using helper methods
- ❑ Duplicated code
  - ☞ factor out the common parts (e.g., using a Template method)
- ❑ Violation of encapsulation
  - ☞ redistribute responsibilities
- ❑ Too much communication between objects (high coupling)
  - ☞ redistribute responsibilities

Various common idioms and patterns can help to improve your design ...

What are Idioms and Patterns?

- ❑ Idioms
  - ☞ Idioms are common programming techniques and conventions.
  - ☞ Idioms may or may not be language-specific.
- ❑ Patterns
  - ☞ Patterns document common solutions to design problems.
  - ☞ Patterns are (intended to be) programming language independent.
- ❑ Libraries
  - ☞ Libraries are collections of functions, procedures or other software components (classes, templates etc.) that can be used in many applications.
- ❑ Frameworks
  - ☞ Frameworks are open libraries that define the generic architecture of an application, and can be extended by adding or deriving new classes.

Frameworks typically make use of many common idioms and design patterns.

Delegation

➤ How does an object share behaviour without inheritance?

✔ *Delegate some of its work to another object*

Inheritance is a common way to extend the behaviour of a class, but can be an inappropriate way to combine features. Delegation reinforces encapsulation by keeping roles and responsibilities distinct.

Example

When a TestSuite is asked to run(), it delegates the work to each of its TestCases.

Consequences

More flexible, less structured than inheritance.

Delegation is one of the most basic object-oriented idioms, and is used by almost all design patterns.

Delegation example

```
public class TestSuite implements Test {
  ...
  /**
   * Runs the tests and collects their result in a TestResult.
   */
  public void run(TestResult result) {
    for(Enumeration e = fTests.elements();
        e.hasMoreElements();)
    {
      if (result.shouldStop() )
        break;
      Test test= (Test) e.nextElement();
      test.run(result);
    }
  }
}
```

Super

➤ How do you extend behaviour inherited from a superclass?

✔ *Overwrite the inherited method, and send a message to "super" in the new method.*

Sometimes you just want to extend inherited behaviour, rather than replace it.

Examples

WrappedStack.top() extends Stack.top() with a pre-condition assertion.

Constructors for subclasses of Exception invoke their superclass constructors.

Consequences

Increases coupling between subclass and superclass: if you change the inheritance structure, super calls may break!

Never use super to invoke a method different than the one being overwritten!

☞ Unnecessarily complex and fragile — use "this" instead

Super example

```
public class WrappedStack extends SimpleWrappedStack {
  ...
  public Object top() throws AssertionException {
    assert(!this.isEmpty());
    return super.top();
  }
  public void pop() throws AssertionException {
    assert(!this.isEmpty());
    super.pop();
  }
}
```

Interface

➤ How do you keep a client of a service independent of classes that provide the service?

✔ *Have the client use the service through an interface rather than a concrete class.*

If a client names a concrete class as a service provider, then only instances of that class or its subclasses can be used in future.

By naming an interface, an instance of any class that implements the interface can be used to provide the service.

Example

Any object may be registered with an Observable if it implements the Observer interface.

Consequences

Interfaces reduce coupling between classes.

They also increase complexity by adding indirection.

Interface example

```
public class GameApplet extends Applet implements Observer
{ ...
  public void update(Observable o, Object arg) {
    Move move = (Move) arg;
    showFeedBack("got an update: " + move);
    _places[move.col][move.row].setMove(move.player);
  }
}
```

Adapter

➤ How do you use a class that provide the right features but the wrong interface?

✔ *Introduce an adapter.*

An adapter converts the interface of a class into another interface clients expect.

Examples

A WrappedStack adapts java.util.Stack, throwing an AssertionException when top() or pop() are called on an empty stack.

An ActionListener converts a call to actionPerformed() to the desired handler method.

Consequences

The client and the adapted object remain independent.

An adapter adds an extra level of indirection.

Also known as Wrapper

Adapter example

```
private Component makeControls() {
  Button again = new Button("New game");
  again.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      showFeedBack("starting new game ...");
      newGame(); // clear the board and bind to a new game
    }
  });
  return again;
}
```

Proxy

➤ How do you hide the complexity of accessing objects that require pre- or post-processing?

✔ *Introduce a proxy to control access to the object.*

Some services require special pre or post-processing. Examples include objects that reside on a remote machine, and those with security restrictions.

A proxy provides the same interface as the object that it controls access to.

Example

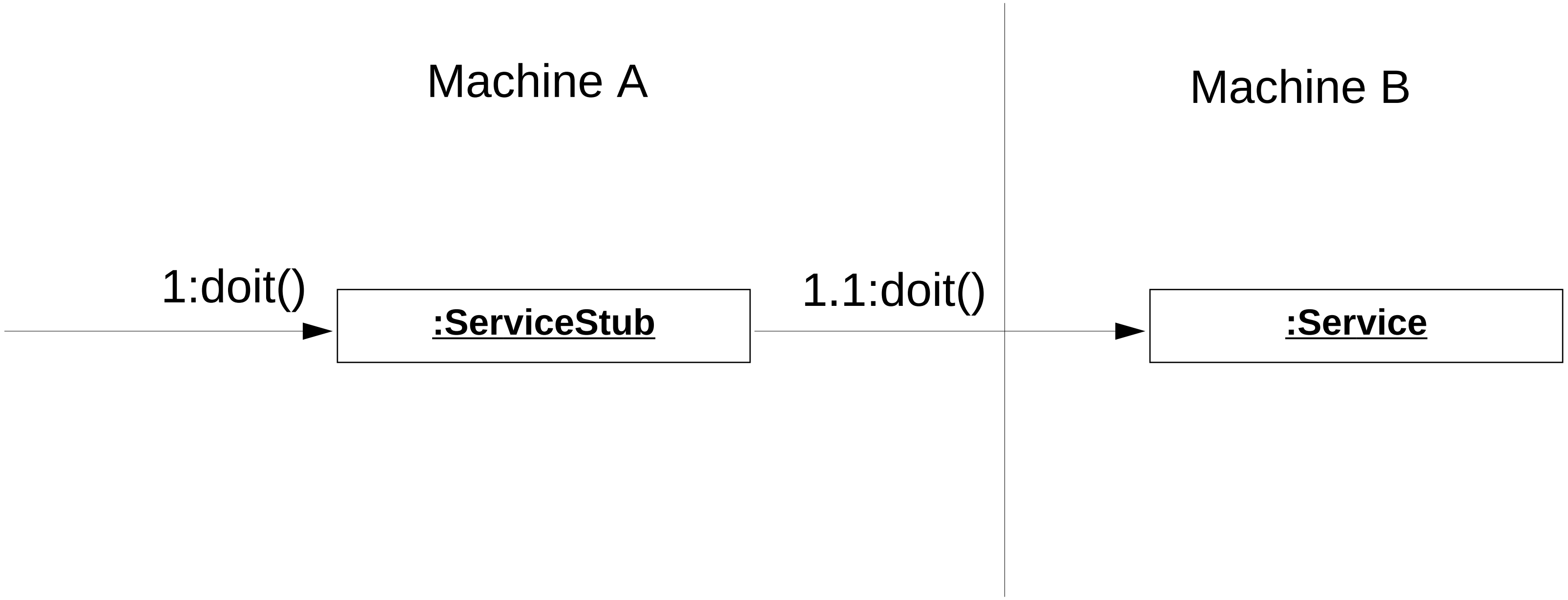
A Java "stub" for a remote object accessed by Remote Method Invocation (RMI).

Consequences

A Proxy decouples clients from servers. A Proxy introduces a level of indirection.

Proxy differs from Adapter in that it does not change the object's interface.

Proxy example



Template Method

➤ How do you implement a generic algorithm, deferring some parts to subclasses?

✔ *Define it as a Template Method.*

A Template Method factors out the common part of similar algorithms, and delegate the rest to *hook methods* that subclasses *may extend*, and *abstract methods* that subclasses *must implement*.

Example

TestCase.runBare() is a template method that calls the hook method setUp().

Consequences

Template methods lead to an *inverted control structure* since a parent classes calls the operations of a subclass and not the other way around.

Template Method is used in most frameworks to allow application programmers to easily extend the functionality of framework classes.

Template method example

Subclasses of TestCase are expected to override hook method setUp() and possibly tearDown() and runTest().

```
public abstract class TestCase implements Test {
  ...
  public void runBare() throws Throwable {
    setUp();
    try {
      runTest();    // by default, look up name
    }               // and run as method
    finally {
      tearDown();
    }
  }
  protected void setUp() { }       // empty by default
  protected void tearDown() { }
}
```

Composite

➤ How do you manage a part-whole hierarchy of objects in a consistent way?

✔ *Define a common interface that both parts and composites implement.*

Typically composite objects will implement their behaviour by delegating to their parts.

Examples

A TestSuite is a composite of TestCases and TestSuites, both of which implement the Test interface.

A Java GUI Container is a composite of GUI Components, and also extends Component.

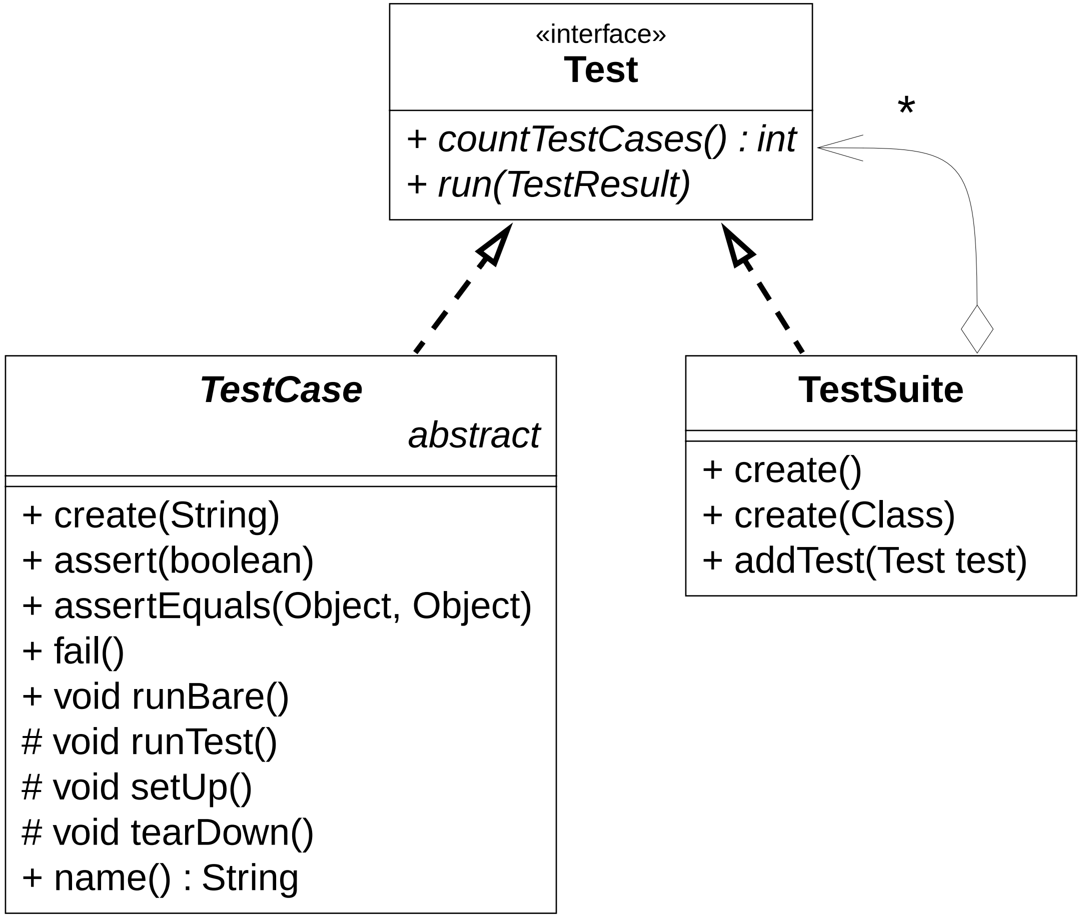
Consequences

Clients can uniformly manipulate parts and wholes.

In a complex hierarchy, it may not be easy to define a common interface that all classes should implement ...

Composite example

A TestSuite is a Test that bundles a set of TestCases and TestSuites.



Observer

➤ How can an object inform arbitrary clients when it changes state?

✔ *Clients implement a common Observer interface and register with the “observable” object; the object notifies its observers when it changes state.*

An observable object *publishes* state change events to its *subscribers*, who must implement a common interface for receiving notification.

Examples

The GameApplet implements java.util.Observable, and registers with a BoardGame.

A Button expects its observers to implement the ActionListener interface.

(see the Interface and Adapter examples)

Consequences

Notification can be slow if there are many observers for an observable, or if observers are themselves observable!

What Problems do Design Patterns Solve?

Patterns document *design experience:*

- ❑ Patterns enable widespread reuse of software architecture
- ❑ Patterns improve communication within and across software development teams
- ❑ Patterns explicitly capture knowledge that experienced developers already understand implicitly
- ❑ Useful patterns arise from practical experience
- ❑ Patterns help ease the transition to object-oriented technology
- ❑ Patterns facilitate training of new developers
- ❑ Patterns help to transcend “programming language-centric” viewpoints

Doug Schmidt, CACM Oct 1995

Summary

You should know the answers to these questions:

- ❑ What’s wrong with long methods? How long should a method be?
- ❑ What’s the difference between a pattern and an idiom?
- ❑ When should you use delegation instead of inheritance?
- ❑ When should you call “super”?
- ❑ How does a Proxy differ from an Adapter?
- ❑ How can a Template Method help to eliminate duplicated code?

Can you answer the following questions?

- ✎ *What idioms do you regularly use when you program? What patterns do you use?*
- ✎ *What is the difference between an interface and an abstract class?*
- ✎ *When should you use an Adapter instead of modifying the interface that doesn’t fit?*
- ✎ *Is it good or bad that java.awt.Component is an abstract class and not an interface?*
- ✎ *Why do the Java libraries use different interfaces for the Observer pattern (java.util.Observer, java.awt.event.ActionListener etc.)?*

12. Common Errors, a few Puzzles

Overview

- ❑ Common errors:
    - ☞ Round-off
    - ☞ == vs. equals()
    - ☞ Forgetting to clone objects
    - ☞ Dangling else
    - ☞ Off-by-1
    - ☞ Terminating loops with an equality test
- ❑ A few Java puzzles ...

Sources

- ❑ Cay Horstmann, *Computing Concepts with Java Essentials*, Wiley, 1998
- ❑ The Java Report, April 1999

and other miscellaneous sources ...

Round-off errors

What does this print?

```
double f = 2e15 + 0.13;
double g = 2e15 + 0.02;

System.err.println(100*(f-g)); // prints 11?
```

Don’t assume that floating point numbers are exact representations of mathematical values!

== versus equals()

When are two objects equal?

```
Object x = new Object();
Object y = new Object();
x == y                // true or false?
x.equals(y)           // true or false?

String s1 = new String(“This is a string”);
String s2 = new String(“This is a string”);
s1 == s2              // true or false?
s1.equals(s2)         // true or false?

int i = 1; int j = 1;
i == j                // true or false?
```

== denotes object equality (but not for primitive types)

equals() denotes object equality by default, but can be overwritten!

Literal Strings

But ... what happens when we compare the two following strings?

```
String s1 = “This is a string”;
String s2 = “This is a string”;
s1 == s2           // true or false?
s1.equals(s2)      // true or false?
```

Literal strings with the same content refer to the same object!

Always use equals() or compareTo() to compare strings!

Forgetting to clone an object

Is “now” really before “later”?

```
Date now = new Date();
Date later = now;
later.setHours(now.getHours() + 1);

if (now.before(later))
  System.out.println("see you later");
else
  System.out.println("see you now");
```

Object variables contain references to objects, not the objects themselves!

If you need a copy of an object, then you should explicitly create it:

```
Date later = new Date(now.getTime());
```

The dangling else problem.

```
public static void checkEven(int n) {
   boolean result = true;
   if (n>=0)
      if ((n%2) == 0)
         System.out.println(n + " is even");
   else
      System.out.println(n + " is negative");
}
```

What is printed when we run these checks?

```
checkEven(-1);
checkEven(0);
checkEven(1);
```

Always use braces to group nested if { } else { } statements!

Off-by-1 errors

The binomial coefficient $\binom{n}{k}$ is $\frac{n}{1} \times \ldots \times \frac{n-k+1}{k}$. Is this a correct implementation?

```
public static int binomial(int n, int k) {
   int bc = 1;
   for (int i=1; i<k; i++)
      bc = bc * (n+1-i) / i;
   return bc;
}
```

To avoid off-by-1 errors

1. *Count the iterations* — do we always do k multiplications? (no)
2. *Check boundary conditions* — do we start with n/1 and finish with (n-k+1)/k? (no)

Off-by-1 errors are among the most common mistakes in implementing algorithms.

Don’t use equality tests to terminate loops!

Don't use =! to test the end of a range. This factorial function won't work for -1 or 0.5!

```
public static int brokenFactorial(int n) {
  int result=1;
  for (int i=0; i!=n; i++)
    result = result*(i+1);
  return result;
}
```

Always use an inequality test to terminate a loop.

Some other common errors

- ❑ Magic numbers
  - ☞ Never use magic numbers; declare constants instead.

- ❑ Forgetting to set a variable in some branch
  - ☞ If you have non-trivial control flow to set a variable, make sure it starts off with a reasonable default value.

- ❑ Underestimating size of data sets
  - ☞ Don’t write programs with arbitrary built-in limits (like line-length); they will break when you least expect it.

- ❑ Leaking encapsulation
  - ☞ Never return a private instance variable! (return e.g., a clone instead)

Puzzle 1

Are private methods inherited? What happens when a subclass overrides inherited private methods used by other, inherited public methods?

```
public class A {
  public void m() { this.p(); }
  private void p() { }
}

public class B extends A {
  private void p() { }
}
```

Which is called? A.p() or B.p()?

```
A b = new B();
b.m();
```

Static and Dynamic Types

Consider:

```
A a = new B();
```

The *static type* of variable a is A — i.e., the statically declared class to which it belongs.

The static type never changes.

The *dynamic type* of a is B — i.e., the class of the object currently bound to a.

The dynamic type may change throughout the program.

```
a = new A();
```

Now the dynamic type is also A!

Puzzle 2

How does Java decide which overloaded method to call when the argument types overlap? Does Java consider the *static type* or the *dynamic type* of the arguments?

```
public class Puzzle2 {

  class A { }
  class B extends A { }

  void m(A a1, A a2) { };
  void m(A a1, B b1) { };
  void m(B b1, A a1) { };
  void m(B b1, B b2) { };
```

```
  public void run() {
    B b = new B();
    A a = b;
    m(a,a);
    m(a,b);
    m(b,a);
    m(b,b);
  }
}
```

The argument objects are the same in each call! Which methods will actually be called?

Puzzle 2 (part II)

What happens if we comment out m(A,A)? m(B,B)? m(A,B)?

In which cases will the example still compile?

Where it does compile, which methods will be called?

Puzzle 3

How does Java use the static type and the dynamic type of the receiver when deciding which method to invoke?

```
public class Puzzle3 {
  public class A { public void m(A a) { } }
  public class B extends A { public void m(B b) { } }
  public void run() {
    B b = new B();
    A a = b;
    a.m(a);
    a.m(b);
    b.m(a);
    b.m(b);
  }
}
```

In which cases will B.m(B) be called?

Puzzle 4

Which takes precedence? Default values or constructor initialization?

```
public class Puzzle4 {
  class C {
    public int i = 100;
    public int j = 100;
    public int k = init();
    public int l = 0;
    C() { i = 0; k = 0; }
    private int init() { j = 0; l = 100; return 100; }
  } ...
  public void run() {
    C c = new C();
    System.out.println("C.i = " + c.i); // 0 or 100?
    System.out.println("C.j = " + c.j); // 0 or 100?
    System.out.println("C.k = " + c.k); // 0 or 100?
    System.out.println("C.l = " + c.l); // 0 or 100?
  }
}
```

Puzzle 4 (part II)

Which takes precedence? Superclass or subclass initialization?

```
public class Puzzle4 { ...
   abstract class A {
      public int j = 100;
      A() { init(100); j = 200; }
      abstract public void init(int value);
   }
   class B extends A {
      public int i = 0;
      public int j = 0 ;
      public void init(int value) { i = value; }
   }
   public void run() { ...
      B b = new B();
      System.out.println("B.i = " + b.i); // 0 or 100?
      System.out.println("B.j = " + b.j); // 0, 100 or 200?
   }
}
```

Puzzle 5

What happens when both the try and the finally clause try to return a value? Which takes precedence?

```
public class Puzzle5 {
  class A {
    public int m() {
      try { return 1; }
      catch (Exception err) { return 2; }
      finally { return 3; }
    }
  }
  public void run() {
    A a = new A();
    System.out.println(a.m());
  }
}
```

What is printed? 1, 2 or 3?

Summary

You should know the answers to these questions:

- ❑ When can you trust floating-point arithmetic?
- ❑ To which “if” does an “else” belong in a nested if statement?
- ❑ How can you avoid off-by-1 errors?
- ❑ Why should you never use equality tests to terminate loops?
- ❑ Are private methods inherited?
- ❑ What are the static and dynamic types of variables?
- ❑ How are they used to dispatch overloaded methods?

Can you answer the following questions?

- ✎ *When is method dispatching ambiguous?*
- ✎ *Is it better to use default values or constructors to initialize variables?*
- ✎ *If both a try clause and its finally clause throw an exception, which exception is really thrown?*