S7041 Programmierung 2

Object-Oriented Programming with Java

Prof. O. Nierstrasz

Sommersemester 2001

Table of Contents

Patterns, Rules and Guidelines

1. P2 — Object-Oriented Programming

Lecturer:	Prof. Oscar Nierstrasz Schützenmattstr. 14/103
Tel:	631.4618
Email:	Oscar.Nierstrasz@iam.unibe.ch
Assistants:	Sander Tichelaar, Frank Buchli, Marc Hugi
WWW:	www.iam.unibe.ch/~scg/Teaching/P2/ (includes full examples)

Principle Texts:

- David Flanagan, *Java in Nutshell: 3d edition*, O'Reilly, 1999.
- James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999
- 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.	03 - 30	Introduction
2.	04 - 06	Design by Contract
	04 - 13	*Good Friday*
3.	04 - 20	Testing and Debugging
4.	04 - 27	Iterative Development
5.	05 - 04	Inheritance and Refactoring
6.	05 - 11	Programming Tools
7.	05 - 18	A Testing Framework
8.	05 - 25	Collections
9.	06 - 01	GUI Construction
10.	06 - 08	Clients and Servers
11.	06 - 15	Guidelines, Idioms and Patterns
12.	06 - 22	Common Errors, a few Puzzles
	06 - 29	*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

...

Goals ...

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* supports *multiple tasks*.

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

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

Problem: compute the total area of a set of geometric shapes

```
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

...

Responsibility-Driven Design ...

- 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

...

Software Quality ...

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

- *Assertions* (pre- and post-conditions, class invariants)
- Disciplined exceptions

Standards

- Protocols, components, libraries, frameworks with standard *interfaces*
- Software *architectures*, design *patterns*

...

How to achieve software quality ...

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

...

Why use OOP? ...

Component reuse

- client/supplier *contracts* can be made *explicit*, simplifying *reuse*

Evolution

- classes and inheritance *limit the impact of changes*

Why Java?

Special characteristics

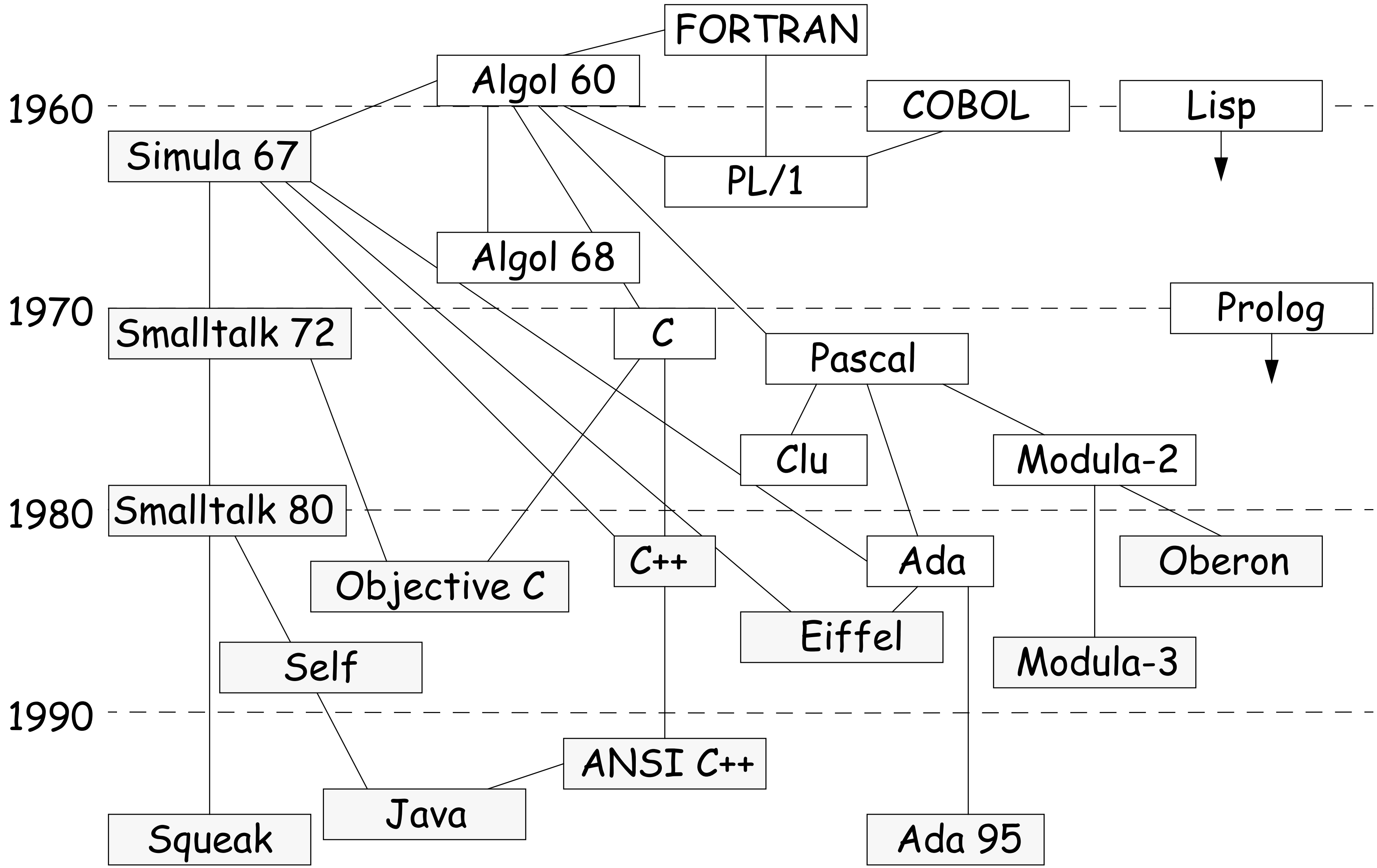
- Resembles C++ *minus the complexity*
- *Clean integration* of many features
- *Dynamically loaded* classes
- Large, *standard* class library

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
Prolog
Smalltalk 72
C
Pascal
Clu
Modula-2
1980
Smalltalk 80
Objective C
C++
Ada
Oberon
Eiffel
Self
Modula-3
1990
ANSI C++
Java
Squeak
Ada 95

What you should know!

- *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 these 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?*
- *Why (when) are case statements bad?*
- *When might it be "all right" to duplicate code?*
- *How do you program classes so they will be "reusable"? Are you sure?*
- *Which is easier to understand — a procedural design or an object-oriented one?*

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.

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

Stacks support two mutating methods: push and pop.

Example: Balancing Parentheses

Problem:

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

Examples:

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

A simple algorithm

Approach:

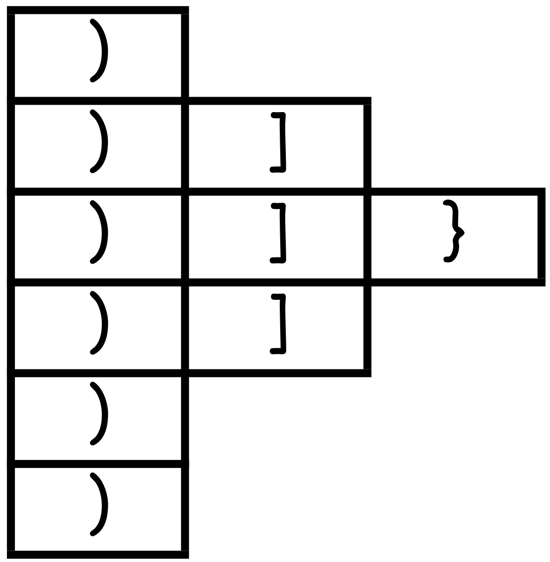
- ❑ when you read a *left* parenthesis, *push* the matching parenthesis on a stack
- ❑ when you read a *right* parenthesis, *compare* it to the value on top of the stack
  - ☞ if they *match*, you *pop* and continue
  - ☞ if they *mismatch*, the expression is *not balanced*
- ❑ if the *stack is empty*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

Stack



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;
  }
```

A declarative algorithm

We implement our algorithm at a *high level of abstraction:*

```
public boolean parenMatch() ... {
  for (int i=0; i<line_.length(); i++) { ...
    if (isLeftParen(c)) { // expect match later
      stack_.push(...(matchingRightParen(c)));
    } else {
      if (isRightParen(c)) { // should equal top
        if (stack_.isEmpty()) { return false; }
        if (stack_.top().equals(new Character(c))) {
          stack_.pop();
        } else { return false; } } } }
  return stack_.isEmpty(); // balanced if empty
}
```

A cluttered algorithm

```
public boolean parenMatch() throws AssertionException {
   for (int i=0; i<line_.length(); i++) {
      char c = line_.charAt(i);
      switch (c) {
      case '{' : stack_.push(new Character('}')); break;
      case '(' : stack_.push(new Character(')')); break;
      case '[' : stack_.push(new Character(']')); break;
      case ']' : case ')' : case '}' :
         if (stack_.isEmpty()) { return false; }
         if (((Character) stack_.top()).charValue() == c) {
            stack_.pop();
         } else { return false; }
         break;
      default : break;
      }
   }
   return stack_.isEmpty();
}
```

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 == '}');
}

...
```

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

Interfaces let us *abstract* from concrete implementations:

```
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;
}
```

➤ How can clients accept multiple implementations of an ADT?

✔ *Make them depend only on 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!

Define your own exception class to *distinguish* your exceptions from any other kind.

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

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*.

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.

...

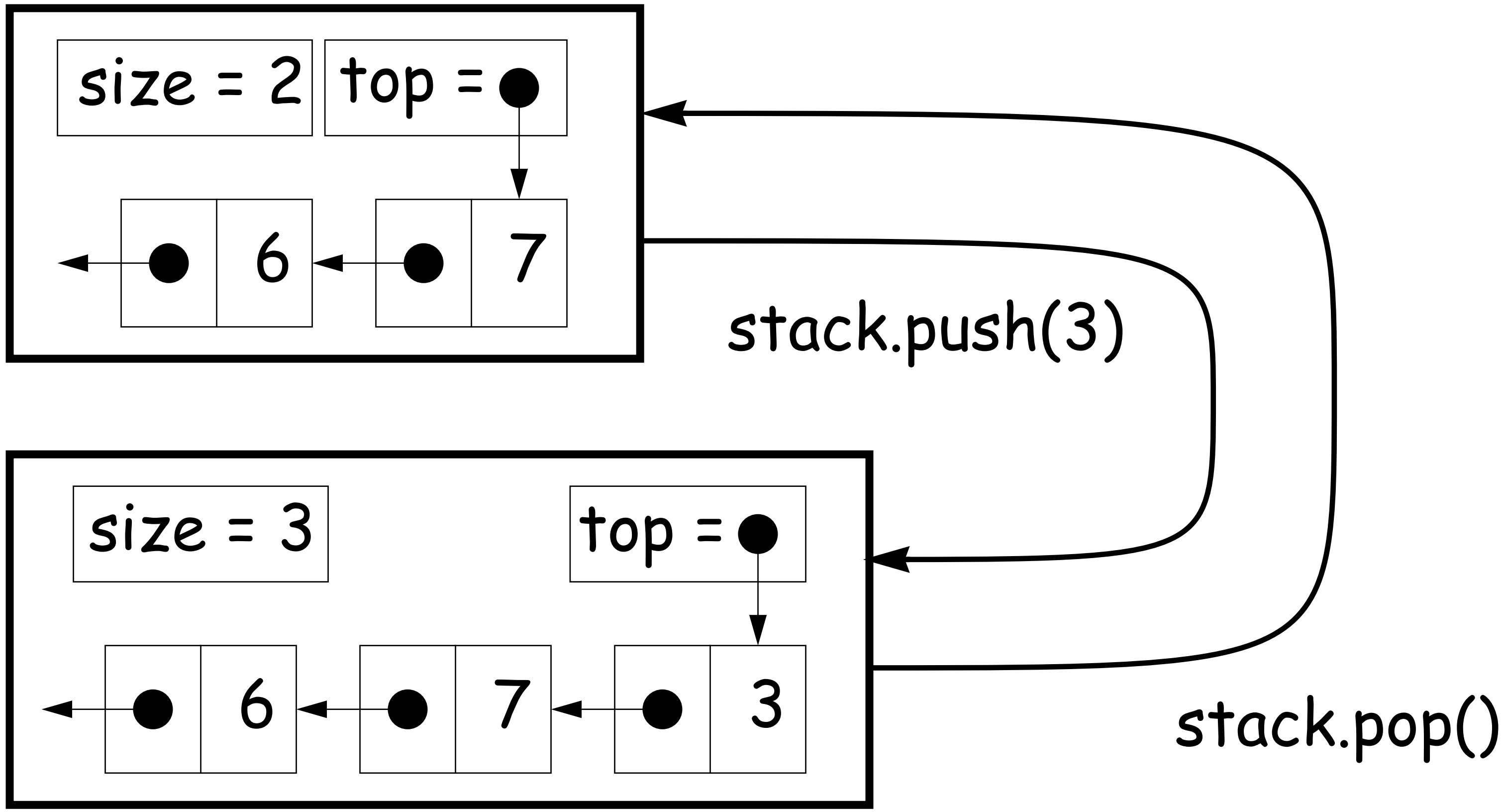
Why are ADTs important? ...

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 by a *linked data structure:*



LinkStack Cells

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

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

Private vs Public instance variables

➤ <u>When should instance variables be public?</u>

✔ *Always make instance variables private or protected.*

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

Naming instance variables

➤ 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.*

LinkStack ADT

The constructor must construct a *valid initial state*:

```
public class LinkStack implements StackInterface {
  ...
  private int size_;
  public LinkStack() {
    // Establishes the invariant.
    top_ = null;
    size_ = 0;
  }
  ...
```

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

Benefits and Obligations

A contract provides *benefits* and *obligations* for both clients and suppliers:

	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

Our Stacks should deliver the following contract:

Operation	*Requires*	*Ensures*
`isEmpty()`	-	*no state change*
`size()`	-	*no state change*
`push(Object item)`	item != null	not empty, size == old size + 1, top == item
`top()`	not empty	*no state change*
`pop()`	not empty	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

Testing Assertions

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

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

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

✔ *Throw an exception.*

Testing Invariants

Every class has its own invariant:

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

Exceptions, failures and errors

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*.

Disciplined Exceptions

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!

Checking pre-conditions

Assert pre-conditions to inform clients when *they* violate the contract.

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

➤ When should you check pre-conditions to methods?

✔ *Always check pre-conditions, raising exceptions if they fail.*

Checking post-conditions

Assert post-conditions and invariants to inform yourself when *you* violate the contract.

```
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());
}
```

➤ When should you check post-conditions?

✔ *Check them whenever the implementation is non-trivial.*

What you should know!

- How can helper methods make an implementation more declarative?
- What is the difference between encapsulation and information hiding?
- 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 these questions?

- Why is strong coupling between clients and suppliers a bad thing?
- When should you call super() in a constructor?
- When should you use an inner class?
- How would you write a general assert() method that works for any class?
- What happens when you pop() an empty java.util.Stack? Is this good or bad?
- What impact do assertions have on performance?
- Can you implement the missing LinkStack methods?

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

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

Testing is always 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!

Caveat: Testing and Correctness

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

Testing a Stack

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

```
static public void testStack(StackInterface stack) {
  try {
    System.out.print("Testing "
        + stack.getClass().getName() + " ... ");
    assert(stack.isEmpty());
... // more tests here ...
    System.out.println("passed all tests!");
  } catch (Exception err) { // NB: any kind!
    err.printStackTrace();
  }
}
```

Build simple test cases

Construct a test case and check the obvious conditions:

```
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);
```

✎ *What other test cases do you need to fully exercise a Stack implementation?*

Check that failures are caught

How do we check that an assertion *fails* when it should?

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

When (not) to use static methods

A <u>*static*</u> method *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!
  - ☞ (counter-)example: *utilities* (java.lang.Math)

...

When (not) to use static variables

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

- 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

We can also implement a (variable) Stack using a (fixed-length) array to store its elements:

```
public class ArrayStack implements StackInterface {
    Object store_ [] = null;// default value
    int capacity_ = 0;      // current size of store
    int size_ = 0;          // number of used slots
...
```

✎ *What would be a suitable class invariant for ArrayStack?*

Handling overflow

Whenever the array runs out of space, the Stack "grows" by allocating a larger array, and copying elements to the new array.

```
public void push(Object item)
                  throws AssertionException
{
  if (size_ == capacity_) {
    grow();
  }
  store_[++size_] = item;   // NB: subtle error!
}
```

✎ *How would you implement the grow() method?*

Checking pre-conditions

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

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!

✎ *Should we also shrink() is the Stack gets too small?*

Testing ArrayStack

When we test our ArrayStack, we get a surprise:

```
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 the *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) ; }
}
```

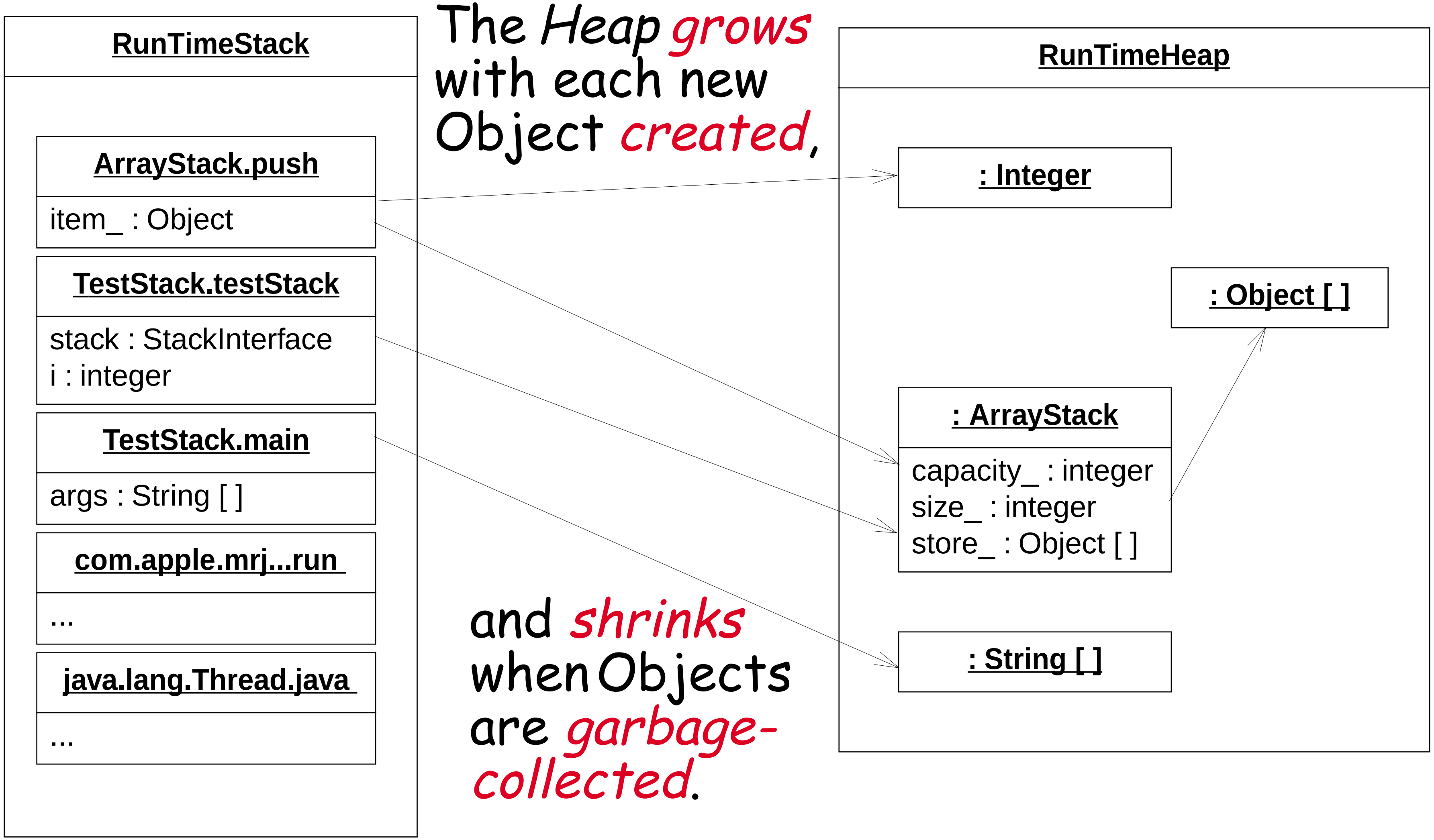
The run-time stack in action ...

A stack frame is *pushed* with each procedure call ...

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

... and *popped* with each return.

The Stack and the Heap



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) ... {
  if (size_ == capacity_) { grow(); }
  store_[size_++] = item; // old size = new size-1
  assert(this.top() == item);
  assert(invariant());
}
```

NB: perhaps it would be clearer to write:

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

java.util.Stack

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 ...

Wrapping Objects

Wrapping is a fundamental programming technique for systems integration.

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

✔ *You wrap it.*

✎ *What are possible disadvantages of wrapping?*

A Wrapped Stack

A wrapper class implements a required interface, by *delegating requests* to an instance of the wrapped class:

```
import java.util.Stack;
public class SimpleWrappedStack
        implements StackInterface
{
  protected Stack stack_;
  public SimpleWrappedStack() {
    stack_ = new Stack();          // wrapped instance
  }
  public boolean isEmpty() {
    return stack_.empty();       // delegation
  }
...
```

A Wrapped Stack ...

```
  public int size() {
    return stack_.size();
  }
  public Object top() throws AssertionException {
    return stack_.peek();
  }
  public void pop() throws AssertionException {
    stack_.pop();
  }
  ... // similar for push()
}
```

✎ *Do you see any flaws with our wrapper class?*

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)
```

✎ *What went wrong?*

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();
  } ...
```

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

```java
import java.util.Date;
public class Timer {                    // Abstract from the
  protected Date startTime_;            // details of timing
  public Timer() {
    this.reset();
  }
  public void reset() {
    startTime_ = new Date();
  }
  public long timeElapsed() {
    return new Date().getTime()
           - startTime_.getTime();
  }
}
```

Sample benchmarks (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
MW JIT	LinkStack	3026	189
	ArrayStack	877	94
	WrappedStack	5927	5318

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

What you should know!

- ✎ What is a regression test? Why is it important?
- ✎ When should you (not) use static methods?
- ✎ 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 these questions?

- Why can't you use tests to demonstrate absence of defects?
- How would you implement ArrayStack.grow()?
- Why doesn't Java allocate objects on the run-time stack?
- 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

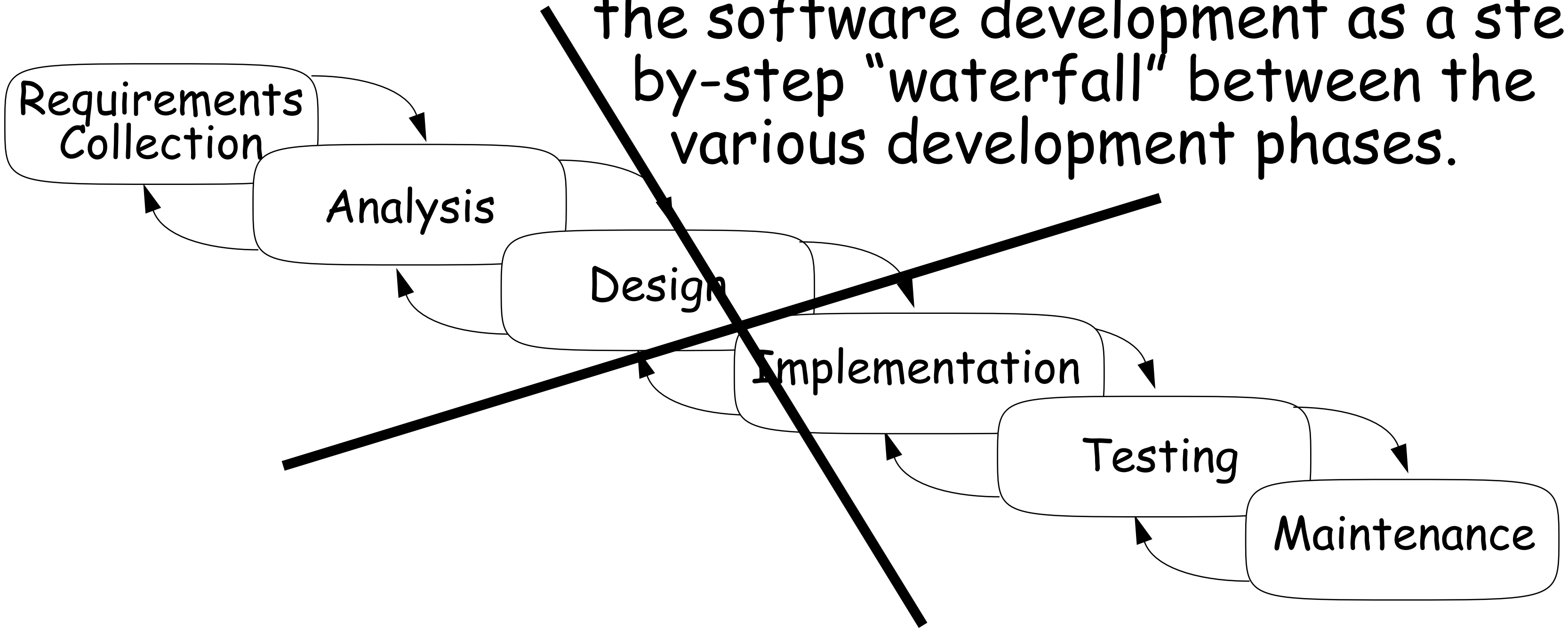
- ❑ Iterative development
- ❑ Responsibility-Driven Design
  - ☞ How to find the objects ...
  - ☞ TicTacToe example ...

Sources

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

The Classical Software Lifecycle

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

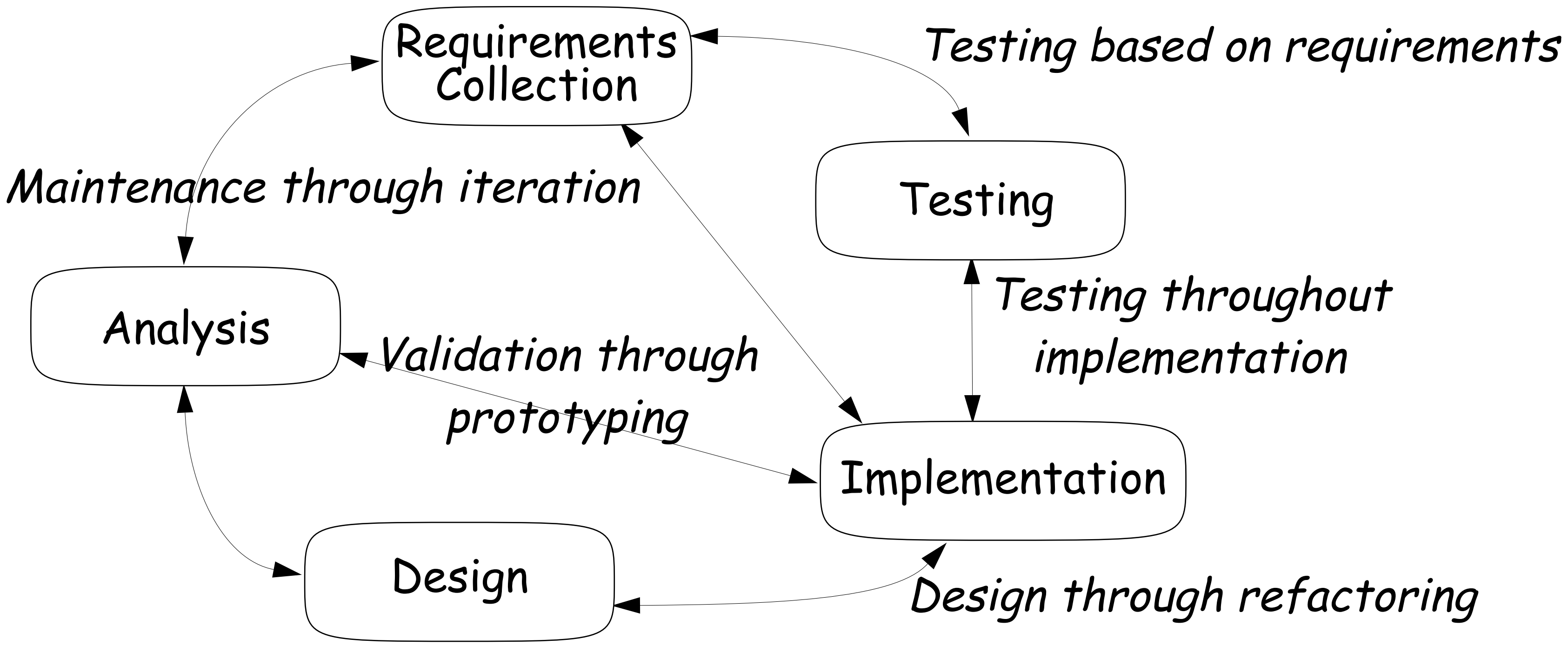


The waterfall model is unrealistic for many reasons, especially:

- 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.



✎ *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).

How to assign responsibility?

Pelrine's Laws:

➤ Which responsibilities should an object accept?

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

➤ How much state should an object expose?

✔ *"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:

> *"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."*
>
> *— Random House Dictionary*

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

Setting 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!

...

Setting Scope ...

An iterative development strategy:

- *limit initial 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

Some objects can be identified from the requirements:

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

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

...

Tic Tac Toe Objects ...

Others can be eliminated:

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

➤ <u>How can you tell when you have the "right" set of objects?</u>

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

Missing Objects

Now we check 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?

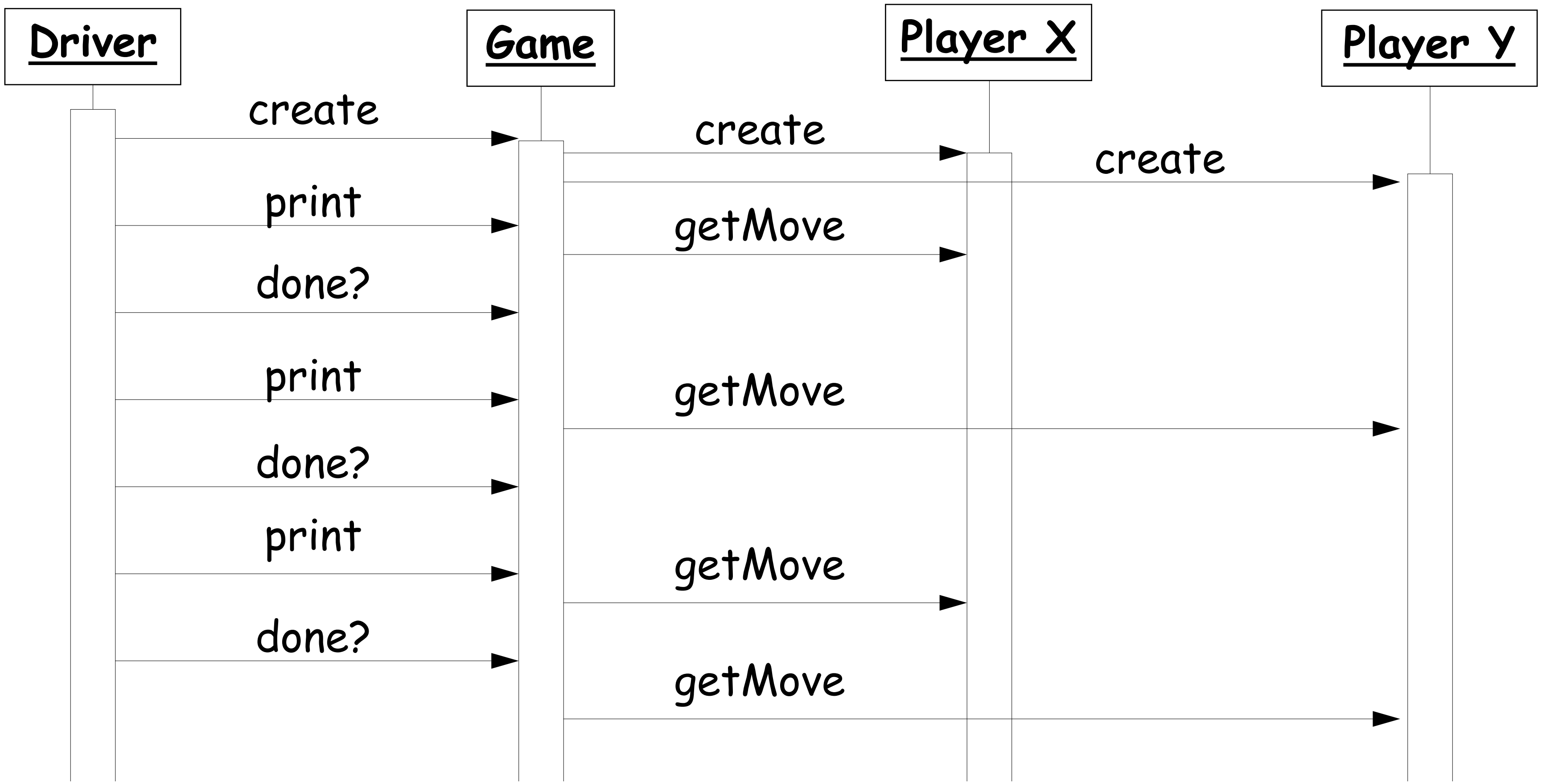
Let us introduce a *Driver* that supervises the Game.

➤ How can you tell if there are objects missing in your design?

✔ *When there are responsibilities left unassigned.*

Scenarios

A *scenario* describes a typical sequence of interactions:



✎ *Are there other equally valid scenarios for this 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,' ');
  }
...
```

Checking pre-conditions

set() and get() translate from chess notation to array indices.

```
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) {
  assert(inRange(col, row));
  return gameState_[col-'a'][row-'1'];
}
private boolean inRange(char col, char row) {
  return (('a'<=col) && (col<='c')
    && ('1'<=row) && (row<='3'));
}
```

Testing the new methods

For now, we 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
```

➤ <u>How do you make an object printable?</u>

✔ *Override Object.toString()*

TicTacToe.toString()

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

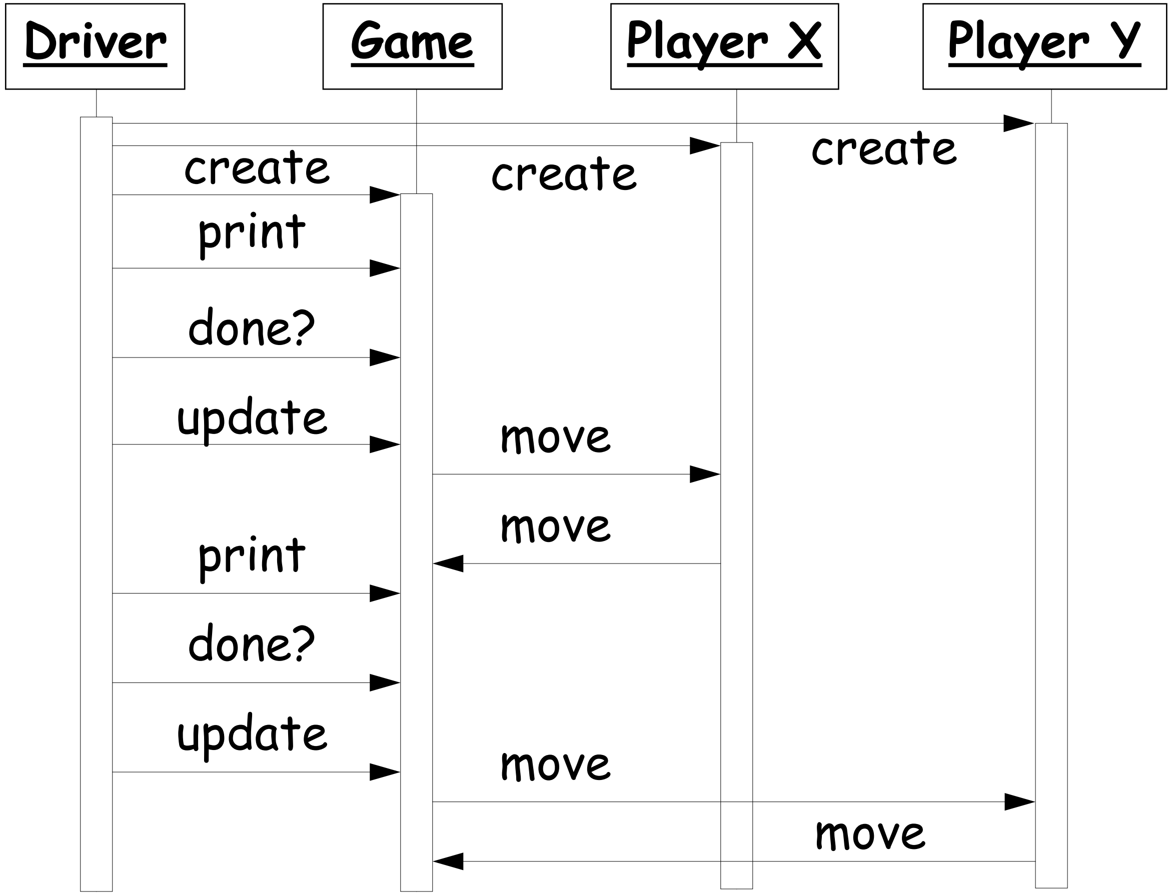
```
public String toString() {
  StringBuffer rep = new StringBuffer();
  for (char row='3'; row>='1'; row--) {
    rep.append(row);
    rep.append("   ");
    for (char col='a'; col <='c'; col++) { ... }
    ...
  }
  rep.append("   a   b   c\n");
  return(rep.toString());
}
```

Refining the interactions

We will want both *real* and *test* Players, so the *Driver* should create them.

Updating the Game and *printing* it should be *separate operations.*

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



Tic Tac Toe Contracts

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 char[][] gameState_;
  private Player winner_ = new Player(); // = nobody
  private Player[] player_;
  private int turn_ = X;               // initial turn
  private int squaresLeft_ = 9;
  static final int X = 0;              // constants
  static final int O = 1;
...
```

Supporting test Players

The Game no longer instantiates the Players, but accepts them as constructor arguments:

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

Invariants

These conditions may 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);
}
```

Note that the Driver may not do this directly!

...

Delegating Responsibilities ...

The Player, in turn, calls the Game's move() method:

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

Small Methods

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;
}
```

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

Accessor Methods

Accessor methods protect clients from changes in implementation:

```
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;
    }
  } ...
```

Code Smells ...

More of the same ...

```
...
  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;
    }
  }
...
```

and yet some more ...

Code Smells ...

```
  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;
  }
}
```

✎ *Duplicated code stinks! How can we clean it up?*

GameDriver

In order to run test games, we separated *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

We use *different constructors* to make real or test Players:

```
public class Player {
  private final char mark_;
  private final BufferedReader in_;
```

A real player reads from the standard input stream:

```
  public Player(char mark) {
    this(mark, new BufferedReader(
                 new InputStreamReader(System.in)
               ));
  }
```

This constructor just calls another one ...

```
...
```

Player constructors ...

But a Player can be constructed that reads its moves from any input buffer:

```
protected Player(char mark, BufferedReader in) {
  mark_ = mark;
  in_ = in;
}
```

This constructor is not intended to be called directly.

```
...
```

Player constructors ...

A test Player gets its input from a String buffer:

```
public Player(char mark, String moves) {
  this(mark, new BufferedReader(
                new StringReader(moves)
              ));
}
```

The default constructor returns a dummy Player representing "nobody"

```
public Player() {
  this(' ');
}
```

Defining test cases

The TestDriver builds games using test Players that represent various 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);
    // ...
  }
...
```

Checking test cases

The TestDriver checks if the results are the expected ones.

```
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 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
```

What you should know!

- *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 these 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
  - ☞ interfaces and abstract classes
- Refactoring
  - ☞ iterative strategies for improving design
- Top-down decomposition
  - ☞ decomposing algorithms to reduce complexity

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.

Inheritance mechanisms

OO languages realize inheritance in different ways:

self	*dynamically* access subclass methods
super	*statically* access overridden, inherited methods
multiple inheritance	inherit features from *multiple superclasses*
abstract classes	*partially defined classes* (to inherit from only)
mixins	build classes from partial *sets of features*
interfaces	*specify* method argument and return types
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

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

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

...

Uses of Inheritance ...

Software reuse:

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

Conceptual hierarchy is important for *analysis*; polymorphism and reuse are more important for *design* and *implementation*.

Note that these three kinds of inheritance can also be exploited separately and independently.

Class Diagrams

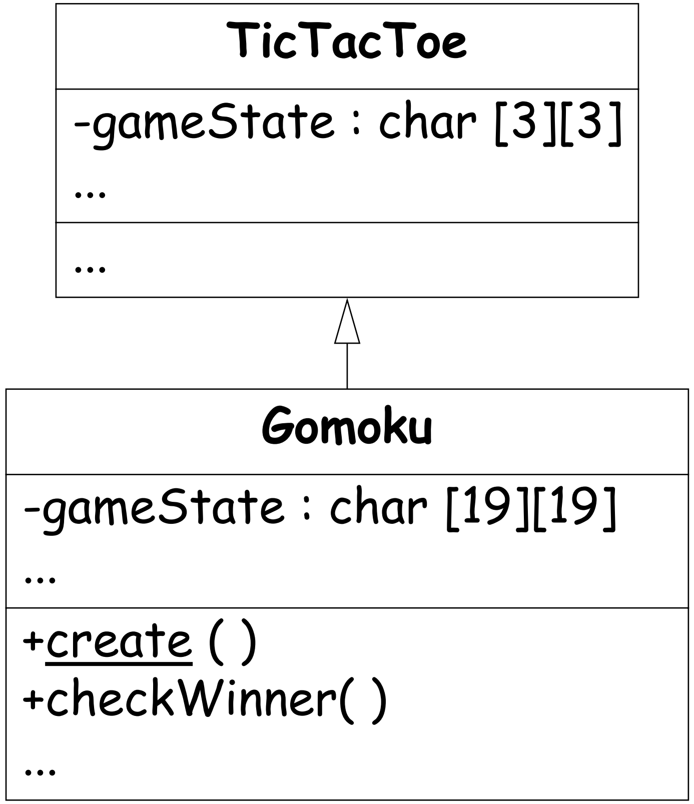
The TicTacToe class currently looks like this:

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

A bad idea ...

Why not simply use inheritance for *incremental modification?*



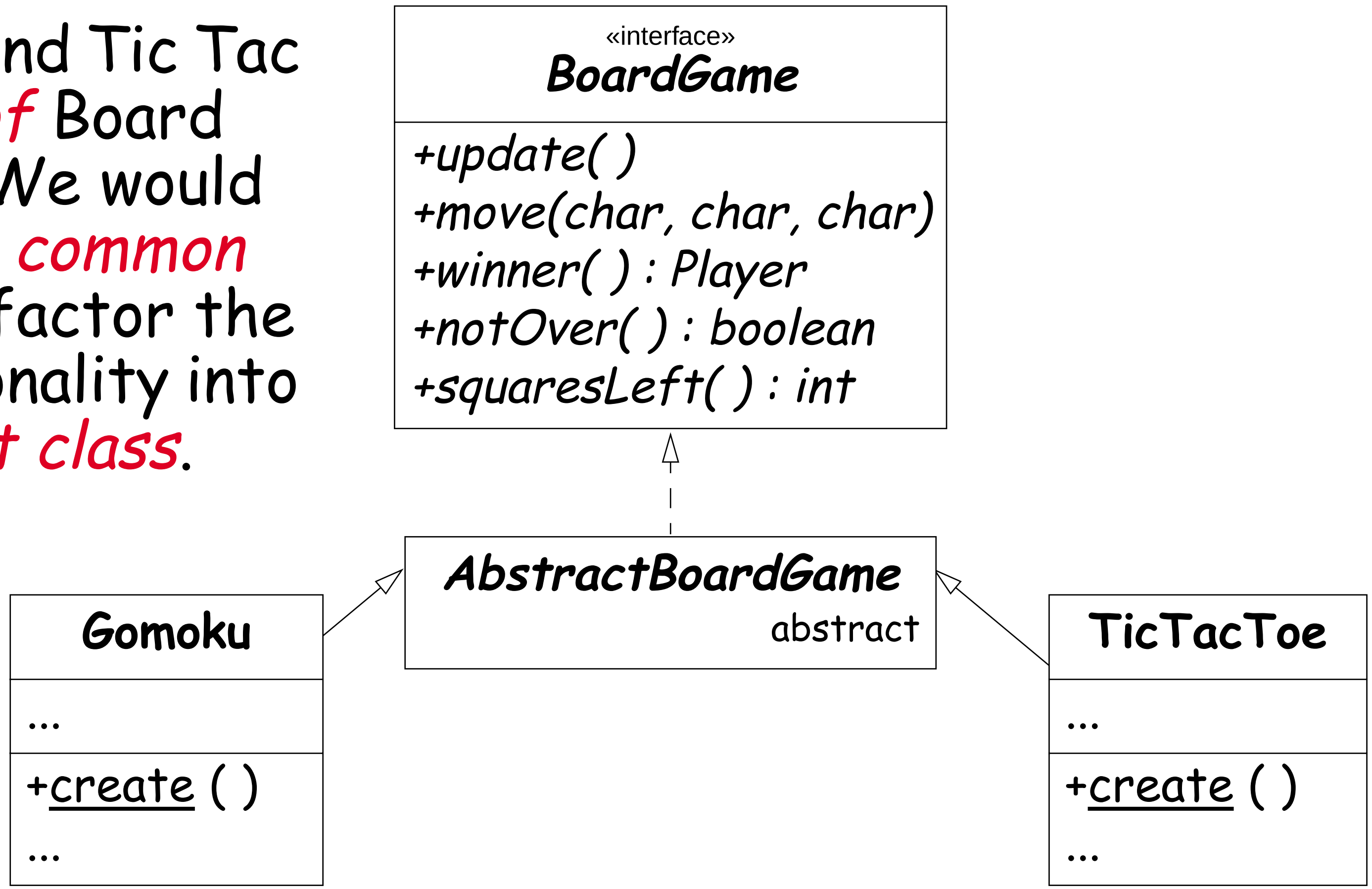
Exploiting inheritance for code reuse *without refactoring* tends to lead to:

- *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!

...

Iterative development strategy ...

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!

➤ <u>When should you run your (regression) tests?</u>

✔ *After every change to the system.*

Version 1.3 (add interface)

We specify the interface both subclasses 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
  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 depend on the BoardGame *interface*:

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

Speak to an interface, not an implementation.

Quiet Testing

Our current TestDriver prints 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,
                            boolean verbose)
{   ...
    if (verbose) {
      System.out.println();
      System.out.println(game);
    ...
}
```

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

Quiet Testing (2)

A more flexible approach is to let the client supply the PrintStream:

```
public static void playGame(BoardGame game,
                            PrintStream out)
{ ...
      out.println(game);
  ...
}
```

The TestDriver can simply send the output to a Null stream:

```
playGame(game, System.out); // normal printing
playGame(game, new NullPrintStream()); // testing
```

NullPrintStream

A Null Object implements an interface with null methods:

```
public class NullPrintStream extends PrintStream {
  NullPrintStream() { super(System.out); }
  public void print() { }
  public void print(Object x) { }
  public void print(String s) { }
  public void println() { }
  public void println(Object x) { }
  public void println(String s) { }
  ...
}
```

Null Objects are useful for eliminating flags and switches.

TicTacToe adaptations

In order to pass responsibility for printing to the GameDriver, a BoardGame 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 provide *common variables* and *methods* for TicTacToe and Gomoku.

```
public abstract class AbstractBoardGame
                        implements BoardGame
{ protected char[][] gameState_;
  protected Player winner_ = new Player();
  protected Player[] player_;
    ...
  protected void set(char col, char row, char mark)
    ...
```

➤ When should a class be declared abstract?

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

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

Refactoring strategies

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 and gradually moved shared code there.

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 ... {
  protected void init(int rows, int cols, int score,
                      Player playerX, Player playerO) { ...
}
```

And call it from the constructors of our subclasses:

```
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's move() method is now radically simplified:

```
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)

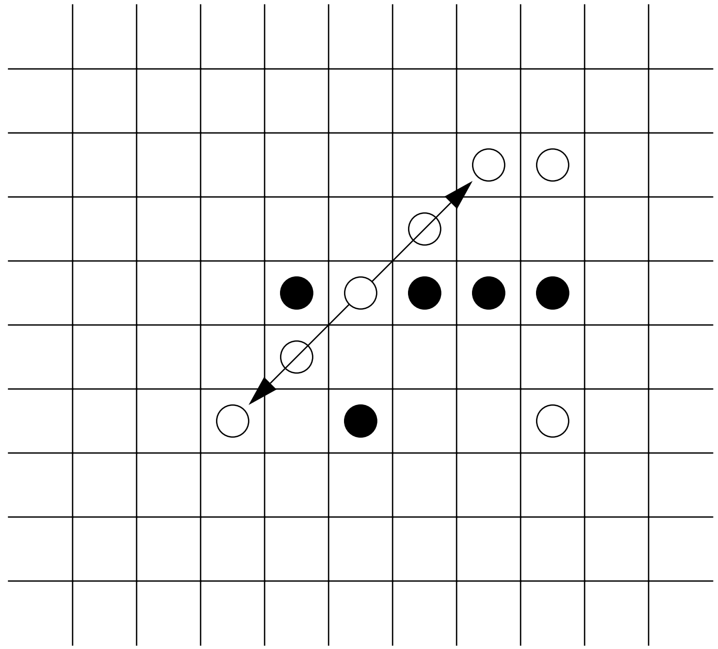
The final steps are:

- 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 exhaustively* for a winning Go-moku score.

We know that *a winning sequence must include the last square marked*. So, it suffices to search in all four directions *starting from that square* to see if we find 5 in a row.



✎ *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*. So let's introduce a *new object* (a Runner) to run and count a Player's pieces.

```
protected void checkWinner(int col, int row)... {
  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_;
  int homeCol_, homeRow_;    // Home col and row
  int col_=0, row_=0;        // Current col & row

  public Runner(BoardGame game, int col, int row)
  {
    game_ = game;
    homeCol_ = col;
    homeRow_ = row;
  }
...
```

Top-down decomposition

Implement algorithms abstractly, introducing helper methods for each abstract step, as you decompose:

```
public int run(int dcol, int drow)
  throws AssertionException
{
  int score = 1;
  this.goHome() ;
  score += this.forwardRun(dcol, drow);
  this.goHome();
  score += this.reverseRun(dcol, drow);
  return score;
}
```

Well-chosen names eliminate the need for most comments!

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;
}
```

More helper methods

Helper methods keep the main algorithm clear and *uncluttered*, and are mostly *trivial to implement*.

```
private int reverseRun(int dcol, int drow) ... {
  return this.forwardRun(-dcol, -drow);
}

private void goHome() {
  col_= homeCol_;
  row_ = homeRow_;
}
```

✎ *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.*

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, Gomoku and TicTacToe could inherit *everything* (except their constructor) from AbstractGameBoard!

What you should know!

- 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?
- Why should tests be silent?

Can you answer these 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?*
- *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:

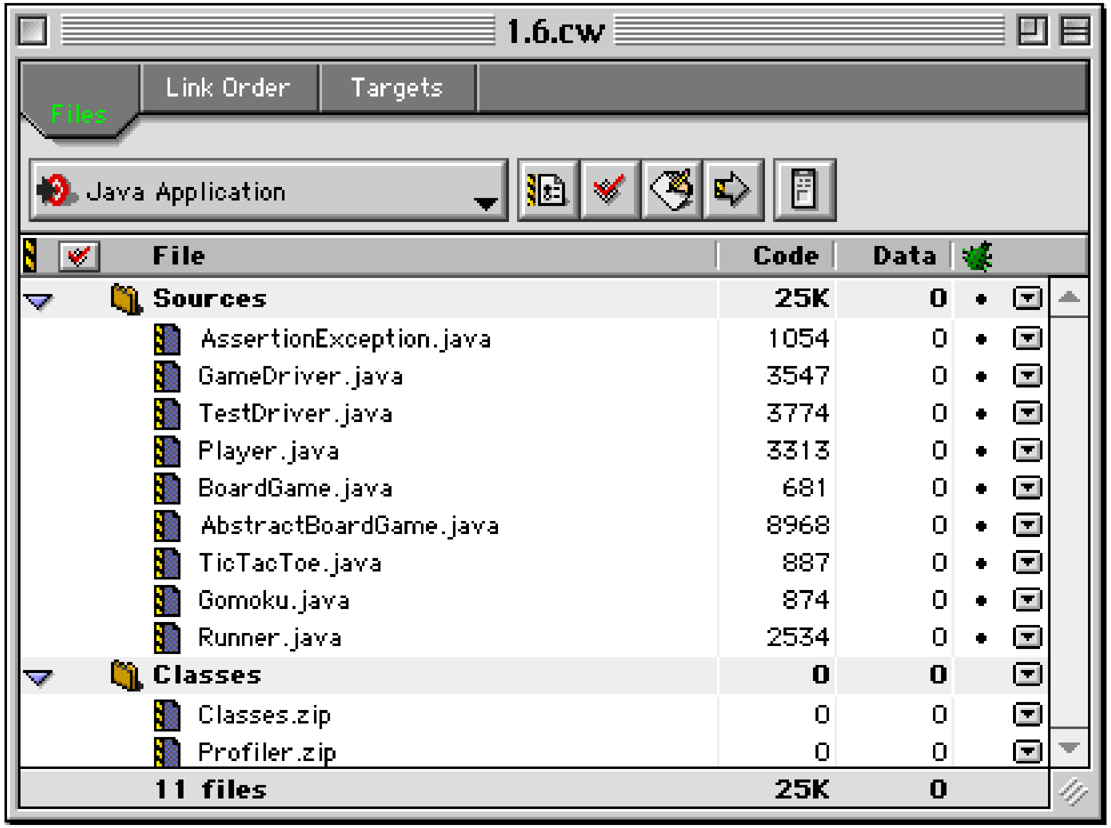
- project manager
- browsers and editors
- compilers and linkers
- make utility
- version control system
- interactive debugger
- profiler
- memory usage monitor
- documentation generator

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

CodeWarrior

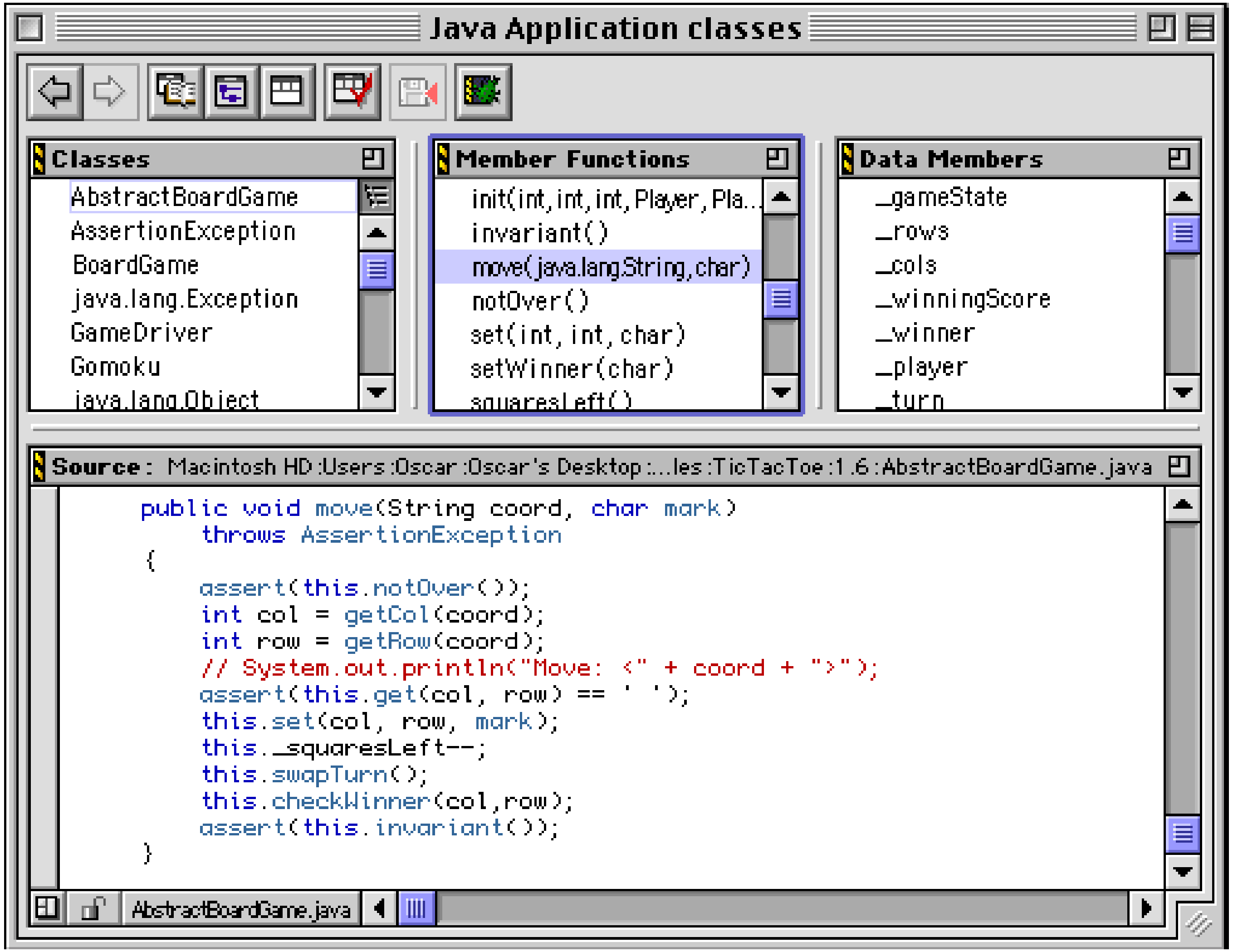
CodeWarrior is a popular IDE for C, C++, Pascal and Java available for MacOS, Windows and Solaris.

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.



CodeWarrior Class Browser

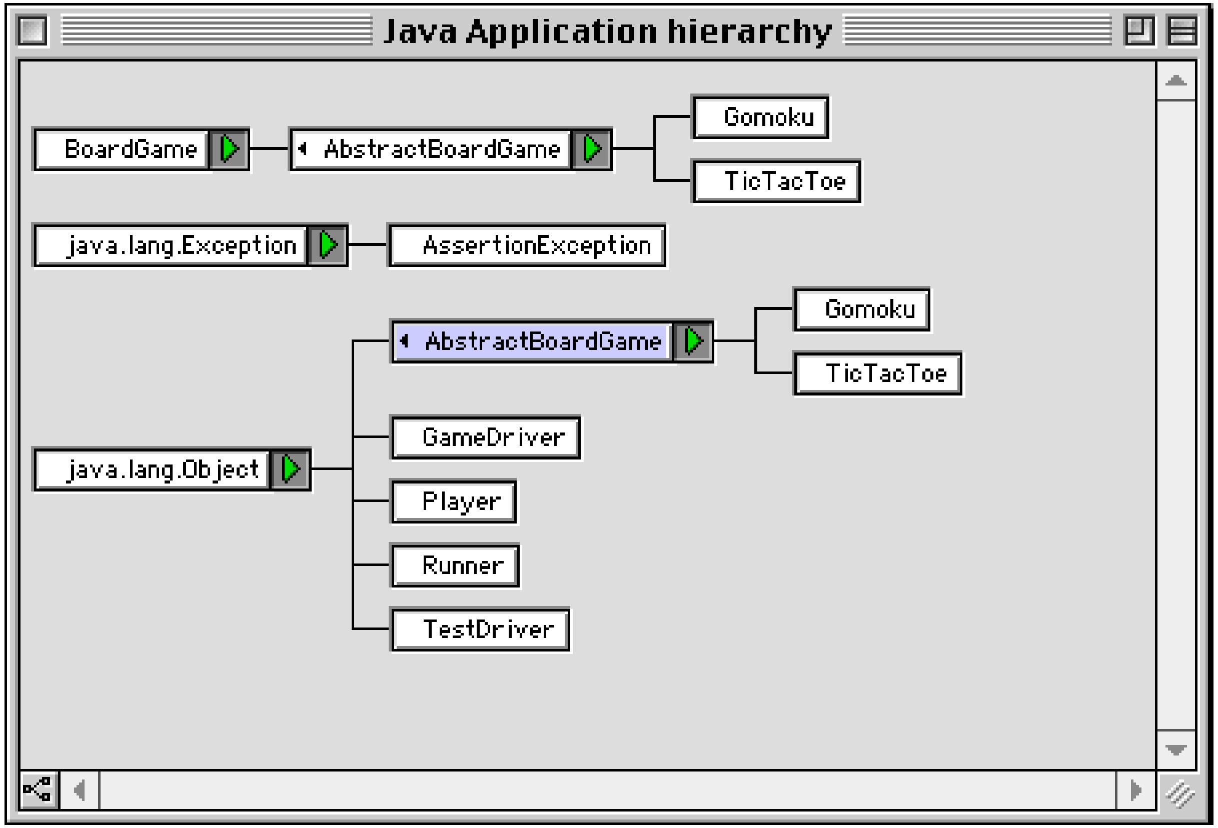
The *Class Browser* provides one way to *navigate* and *edit* project files ...



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!



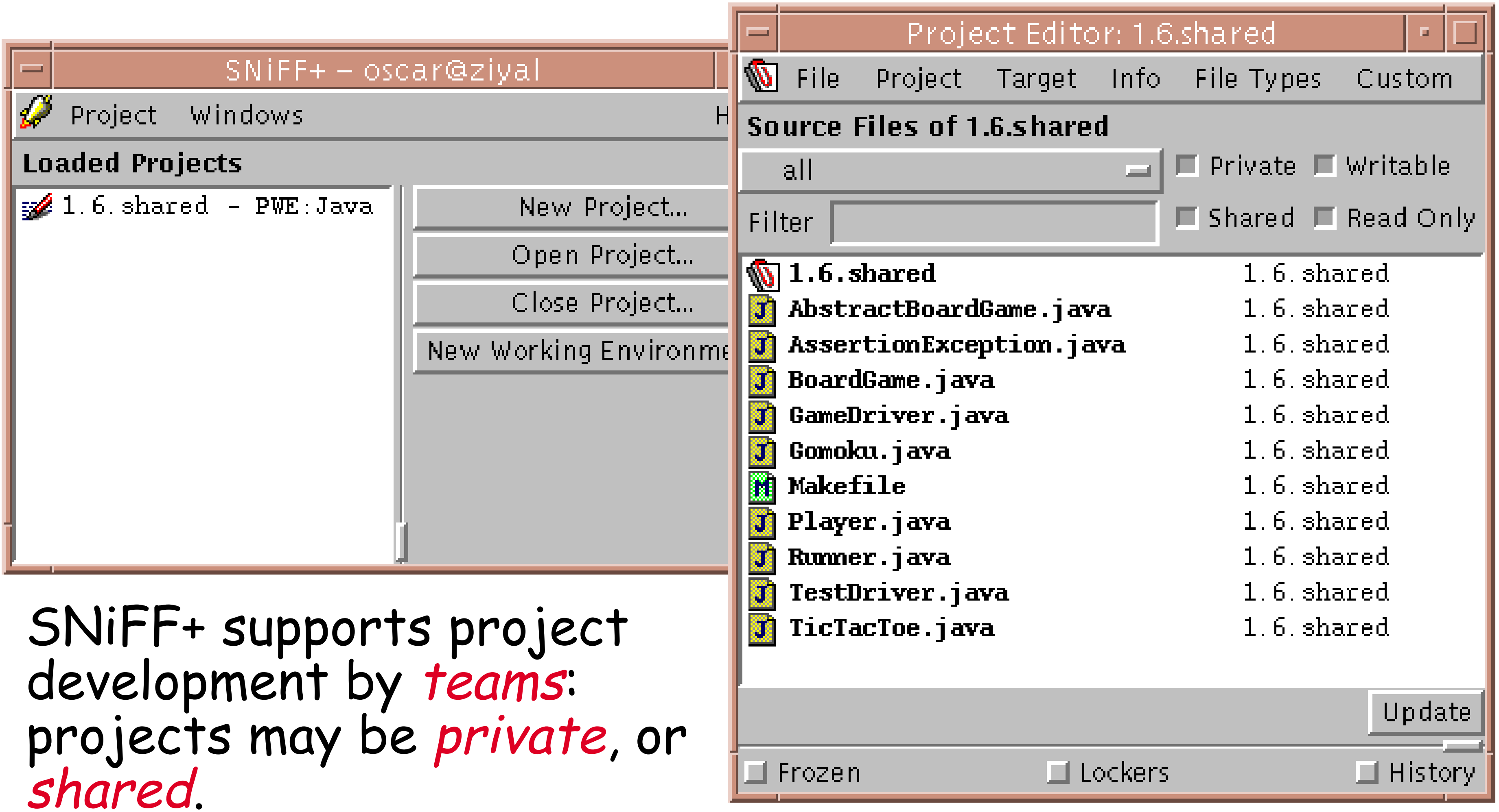
SNiFF+

SNiFF+ is an IDE for C++, Java, Python and 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)

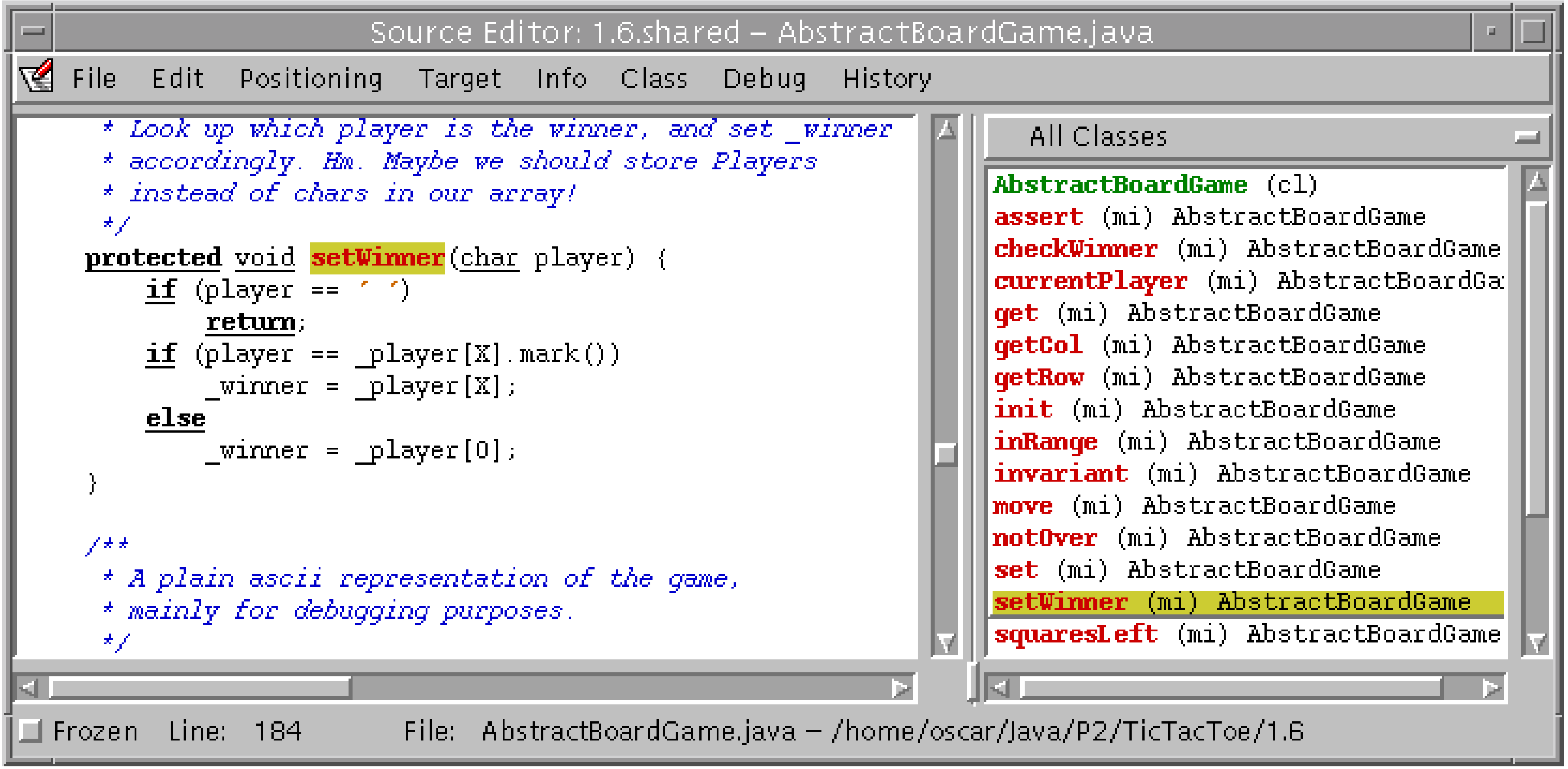
SNiFF+ is an *open IDE*, allowing different compilers, debuggers, etc. to be plugged in.

SNiFF+ Project Editor

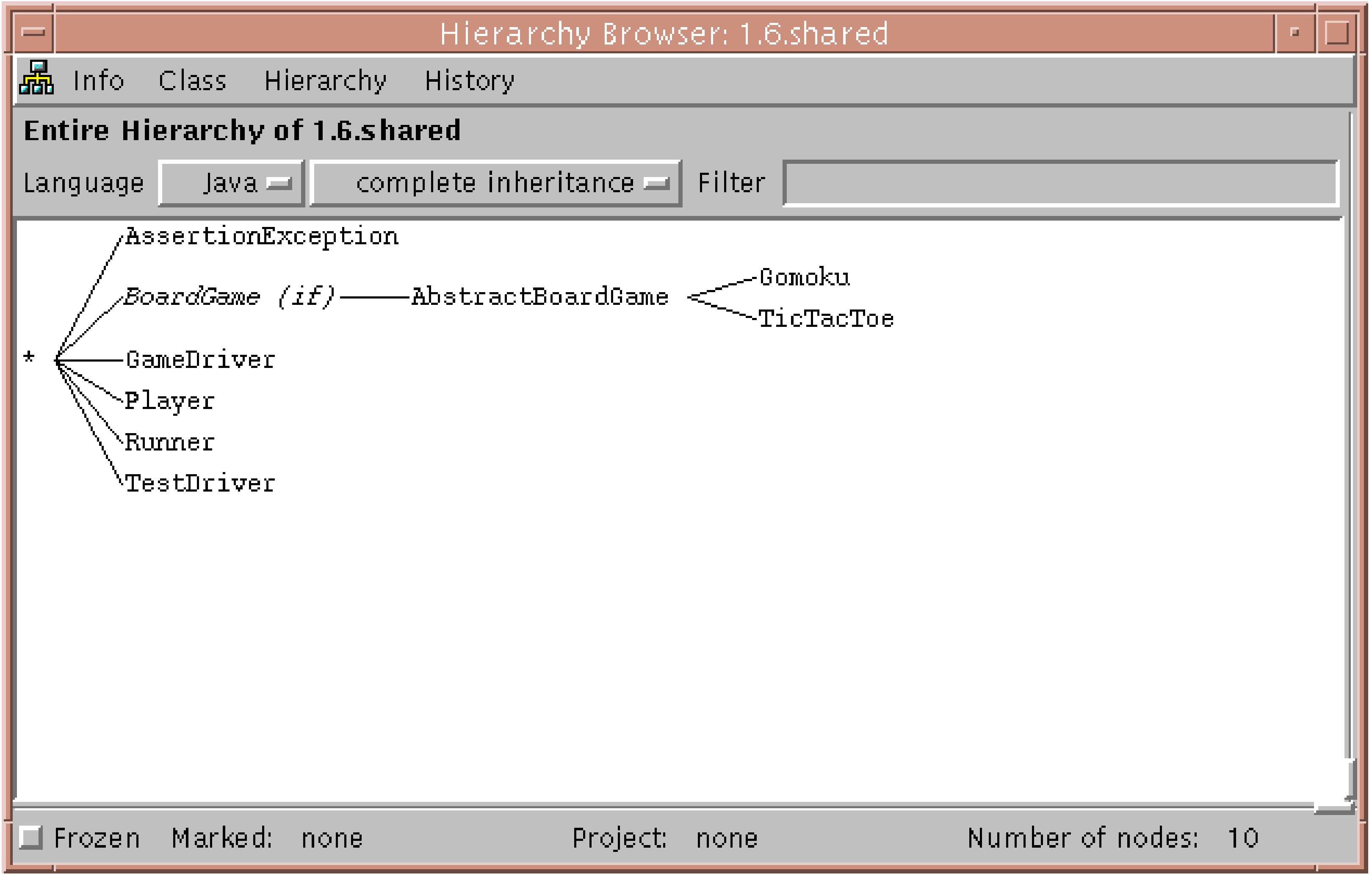


SNiFF+ supports project development by *teams*: projects may be *private*, or *shared*.

SNiFF+ Source Editor



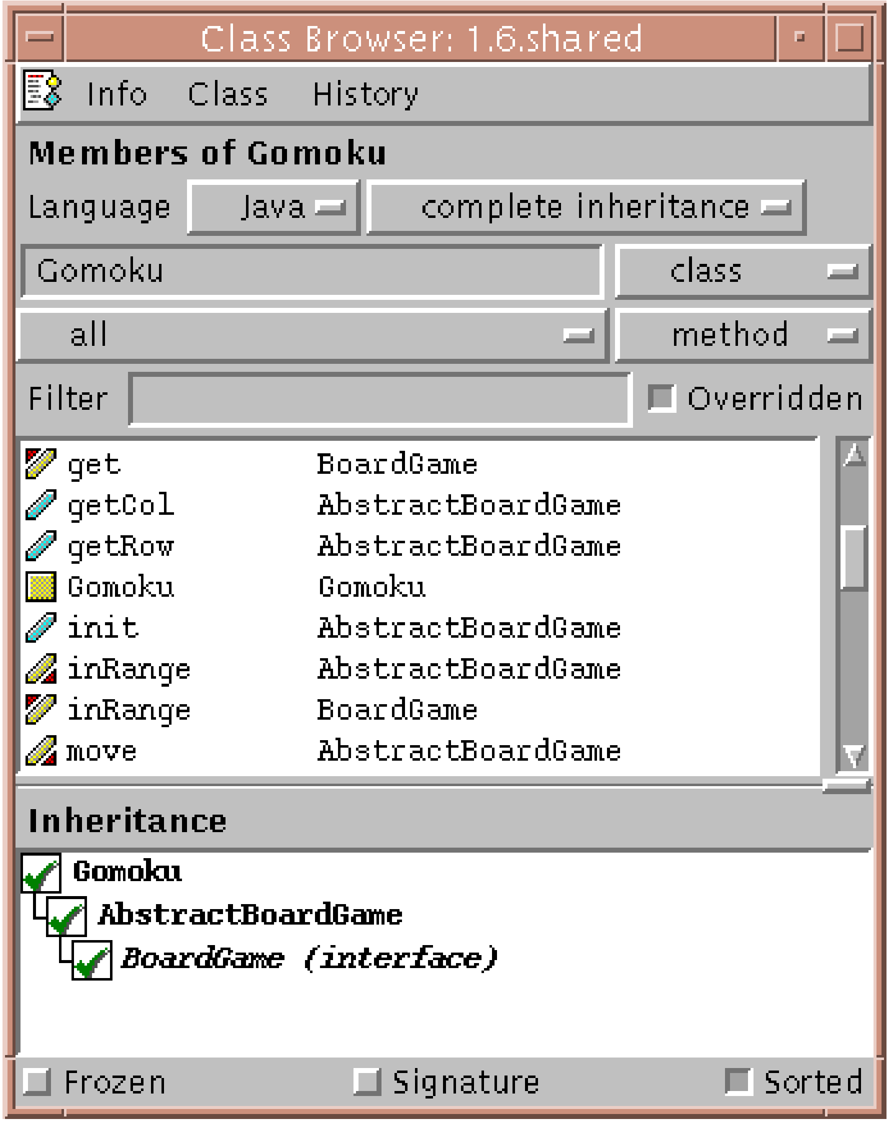
SNiFF+ Hierarchy Browser



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).



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")

Using Debuggers

Interactive debuggers are available for most mature programming languages.

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

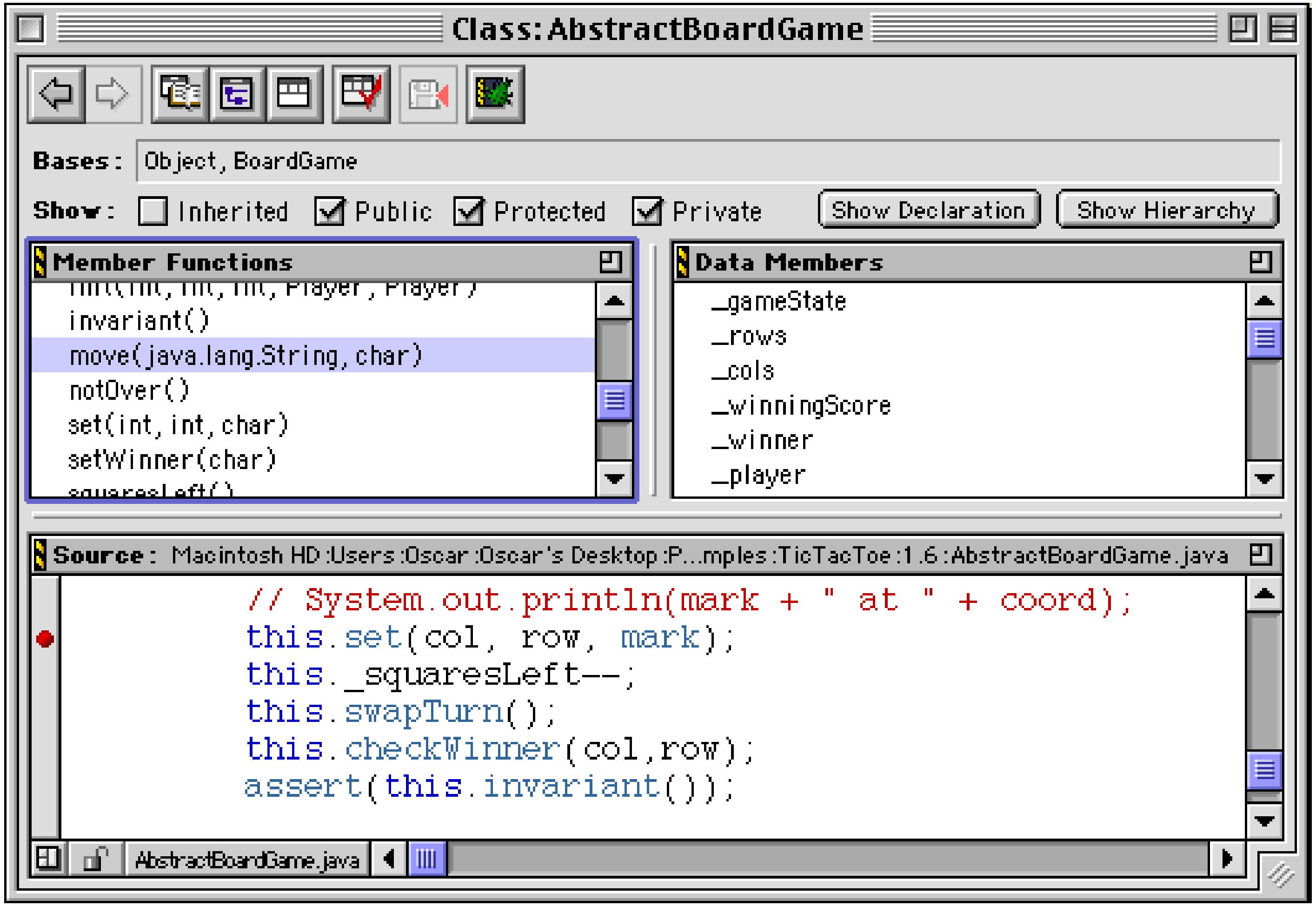
➤ When should you use a debugger?

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

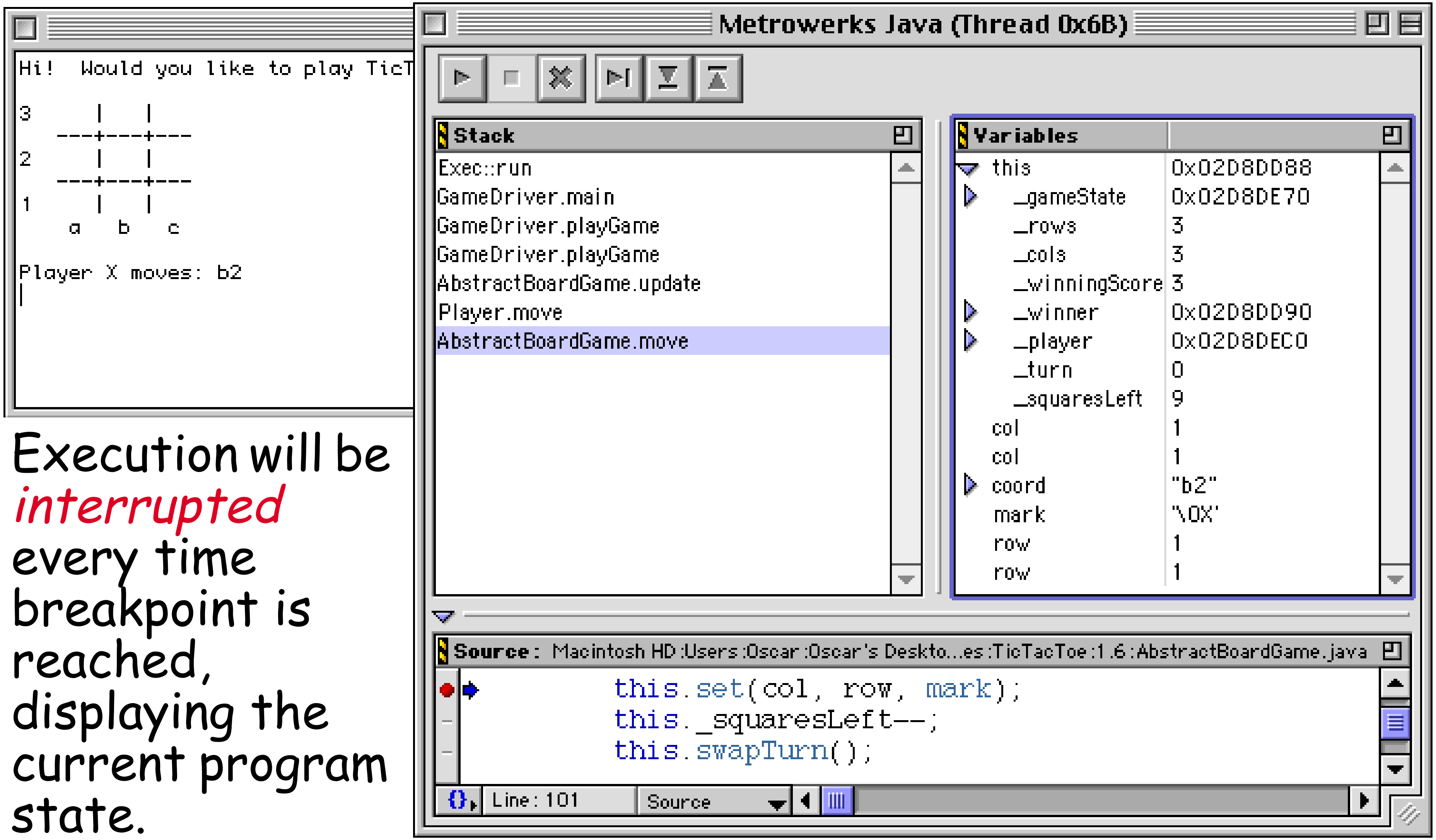
NB: debuggers are object code specific, so can only be used with programs compiled with compilers generating compatible object files.

Setting Breakpoints

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



Debugging



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 probe the program state
  - ☞ print the state only when an assertion fails
- ❑ After every modification, do *regression testing!*

...

Debugging Strategy ...

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!*

All software bugs are a matter of false assumptions.

If you *make your assumptions explicit*, you will find and stamp out your bugs.

Version Control Systems

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.cvshome.org)

Version Control

Version control *enables* you to make *radical changes* to a software system, with the *assurance* that *you can always go back* to the last working version.

➤ <u>When should you use a version control system?</u>

✔ *Use it whenever you have one available, for even the smallest project!*

Version control is as important as testing in iterative development!

RCS command overview

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	*Merge changes* from two files into a third
rcsdiff	Report *differences* between revisions
rcsmerge	*Merge 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; leave a read-only copy
ci -l file     # check in file; leave a locked 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)

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 <u>*profiler*</u> (e.g., java -prof) tells you where a terminated 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

Caveat: the technical details vary from compiler to compiler

...

Using Profilers

➤ <u>When should you use a profiler?</u>

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

➤ <u>How early should you start worrying about performance?</u>

✔ *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!

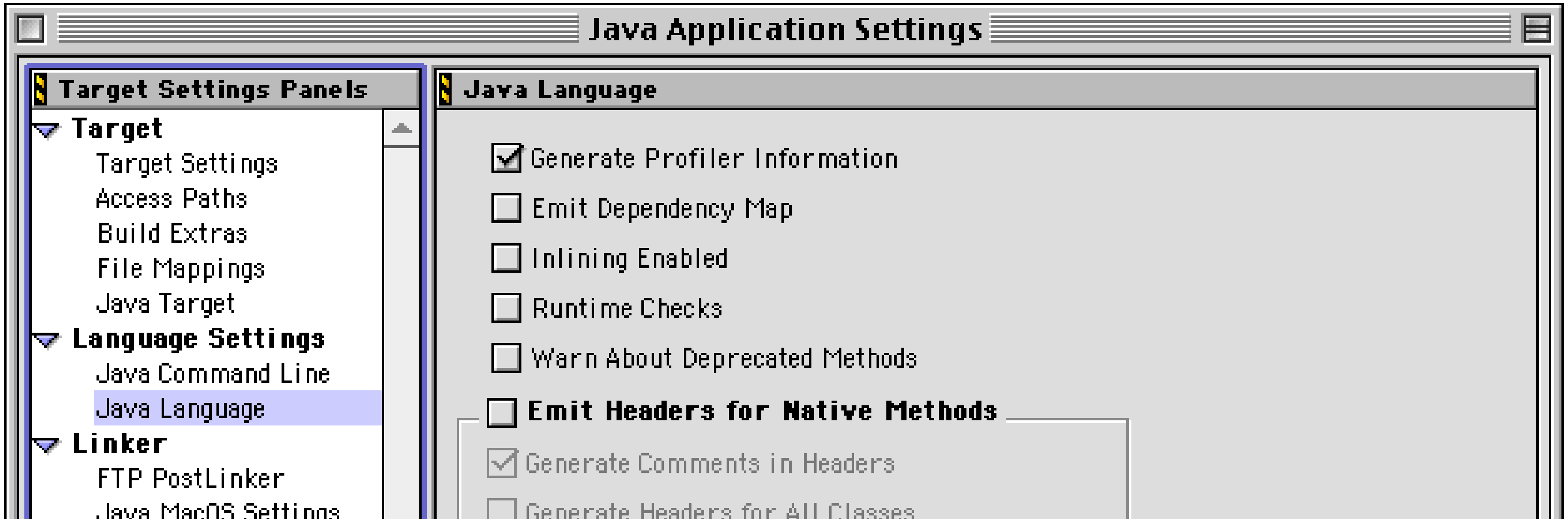
Profiling with CodeWarrior

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();
  } ...
```

Profiling with CodeWarrior ...

and turn on profiling:



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

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 */.

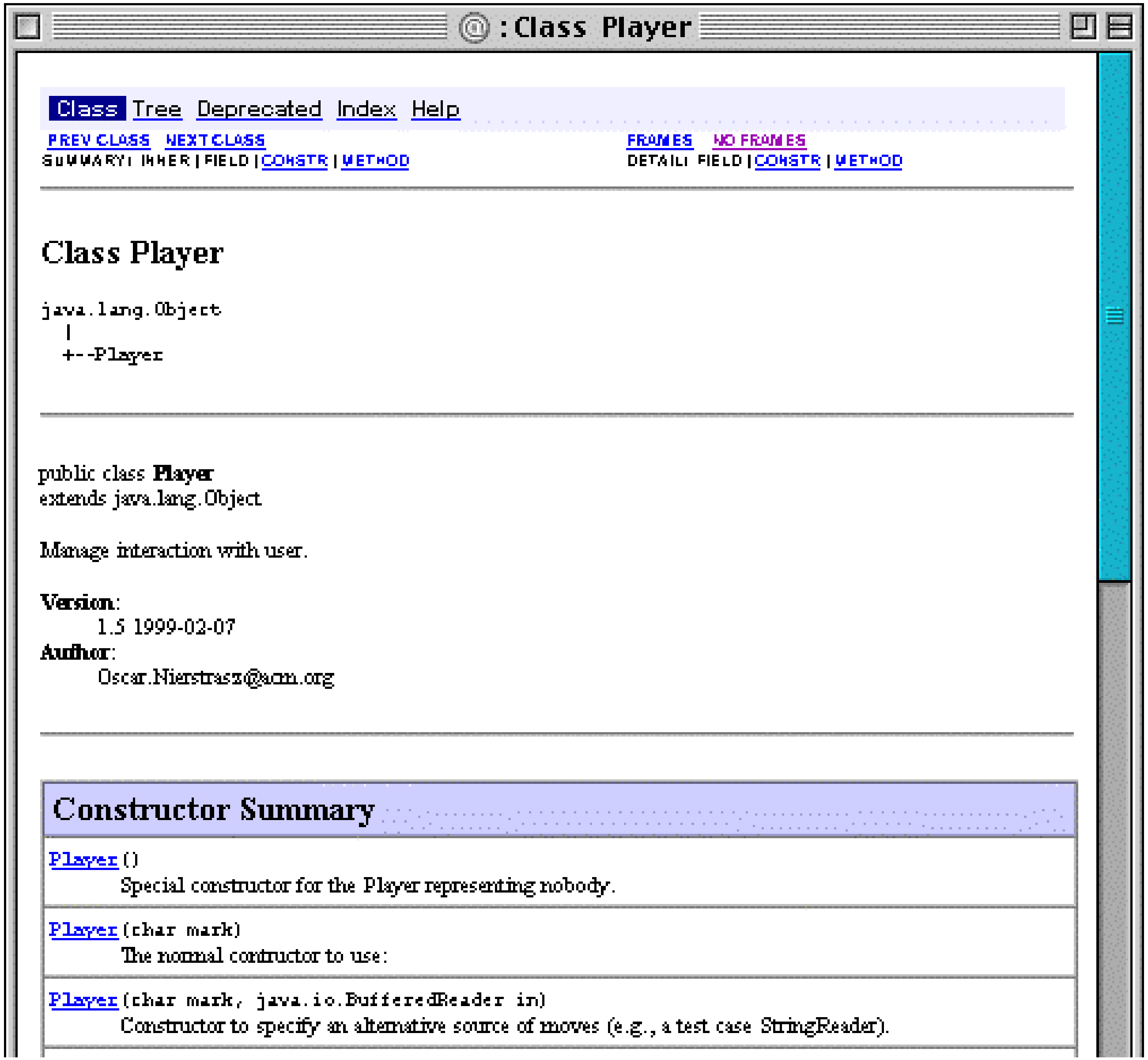
Comments may contain *special tag values* (e.g., @author) and (some) HTML tags.

Javadoc input

```
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!



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"

...

Other tools ...

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

What you should know!

- 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 these 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 3.5 documentation (from www.junit.org)

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 small steps, and *run the (regression) tests after each step*.

  Fix what's broken before proceeding.

...

Testing Practices ...

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*

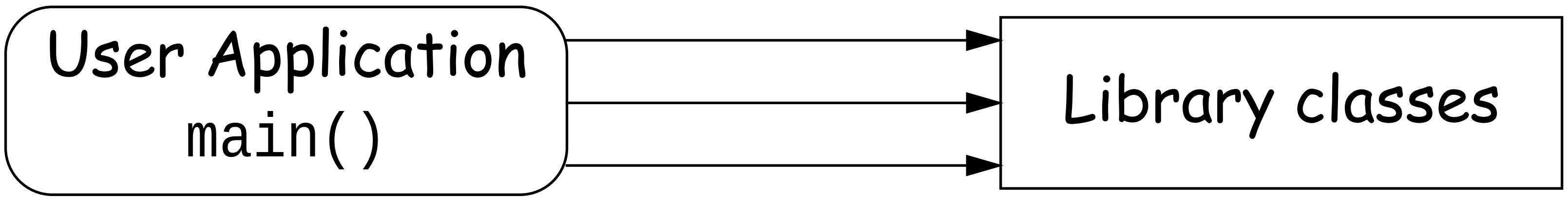
JUnit distinguishes between *failures* and *errors*:

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

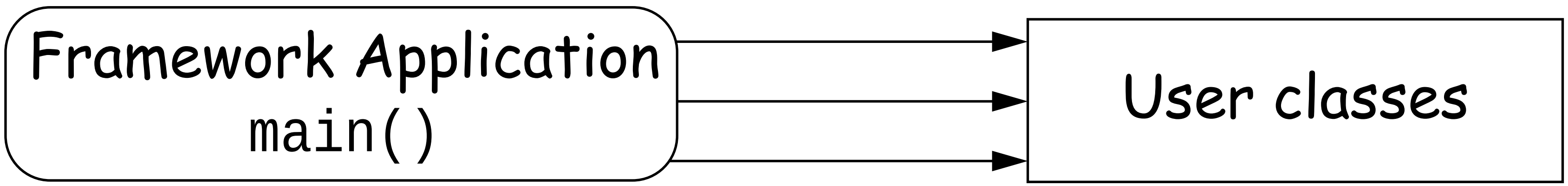
NB: this is not the same distinction made by Meyer!

Frameworks vs. Libraries

In traditional application architectures, *user code* makes use of *library functionality* in the form of procedures or classes:

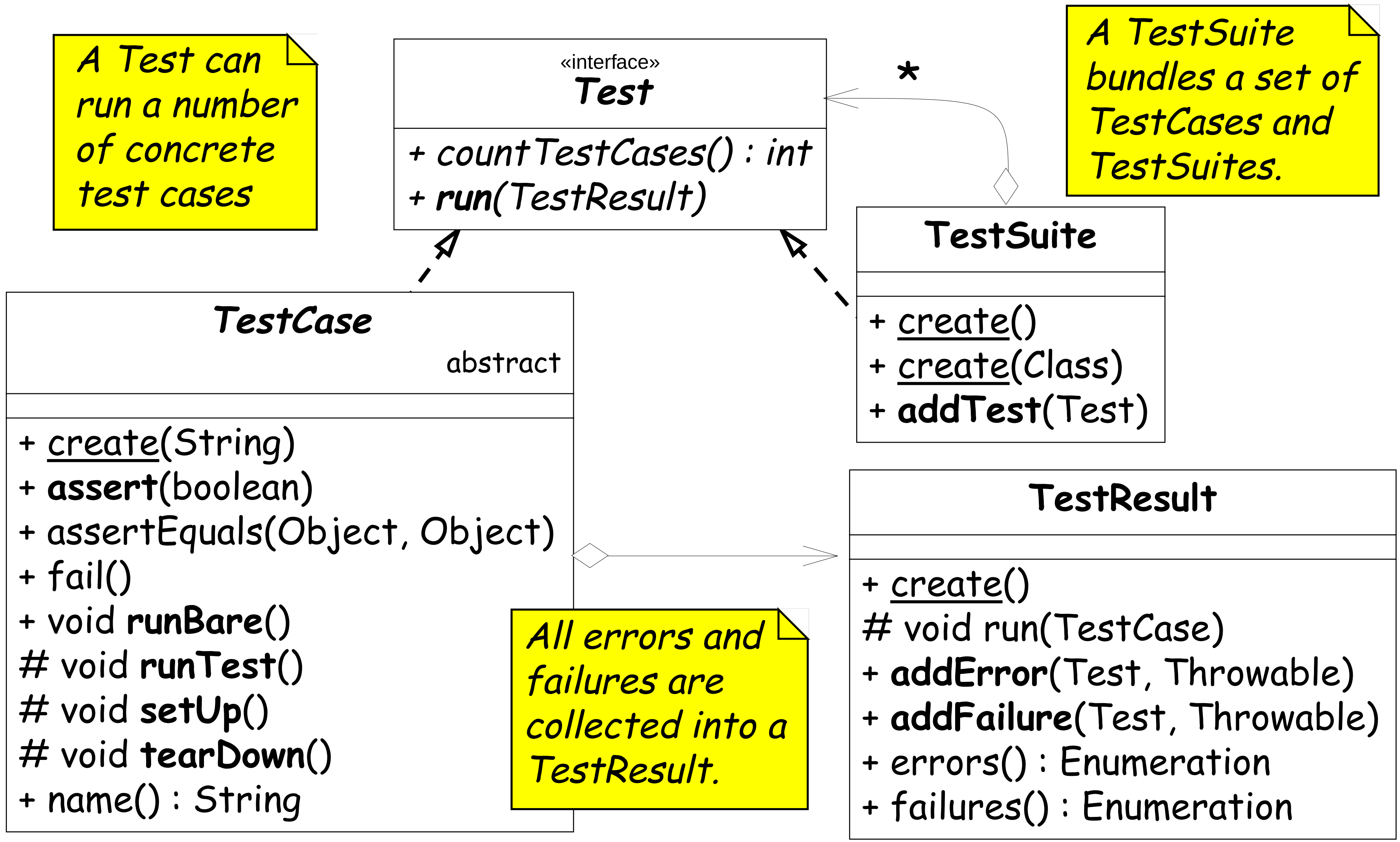


A framework *reverses* the usual relationship between generic and application code. Frameworks provide *both* generic functionality *and* application architecture:

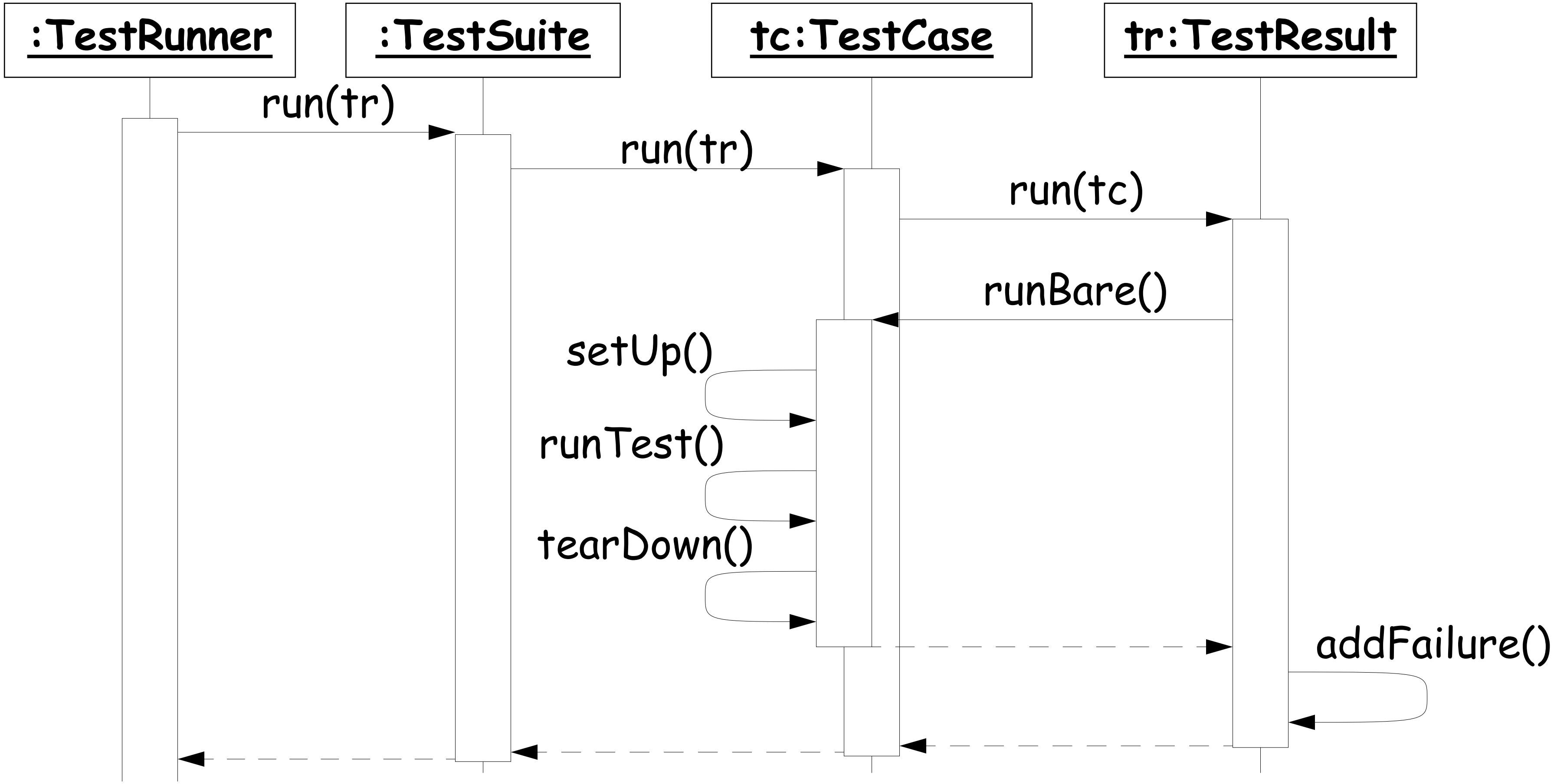


Essentially, a framework says: "Don't call me — I'll call you."

The JUnit Framework



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:

```
public class Money {
  ...
  public Money add(Money m) {
    return new Money(...);
  }
  ...
}
```

Money
- fAmount : int - fCurrency : String
+ <u>create</u>(int, String) + amount() : int + currency() : String + add(Money) : Money + equals(Object) : boolean + toString() : String

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 (the *fixture*):

```
import junit.framework.*;
public class MoneyTest extends TestCase {
  private Money f12CHF;
  private Money f14CHF;
  public MoneyTest(String name) { super(name); }

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

Some basic tests

We define methods to test what we expect to be true ...

```
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);
  assertEquals(expected, result);
}
```

Building a Test Suite

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

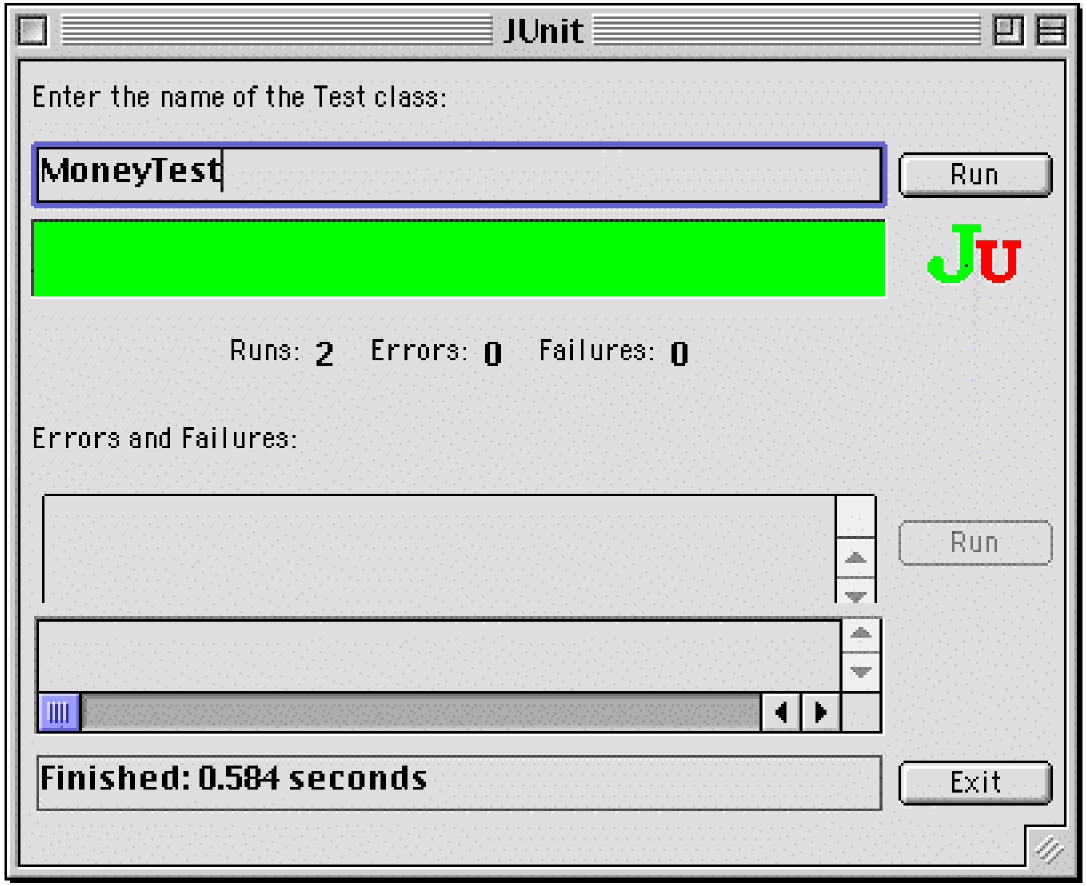
```
public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(new MoneyTest("testEquals"));
  suite.addTest(new MoneyTest("testSimpleAdd"));
  return suite;
}
```

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 *several instances* of Money:

MoneyBag
- fMonies : HashTable
+ create(Money, Money) + create(Money [ ]) - appendMoney(Money) + toString() : String

...

MoneyBags ...

```
class MoneyBag {
  private Hashtable fMonies = new Hashtable(5);
  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);
  }
}
```

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)å;
    fMB2 = new MoneyBag(f14CHF, f21USD);
  }
```

Testing MoneyBags (II)

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

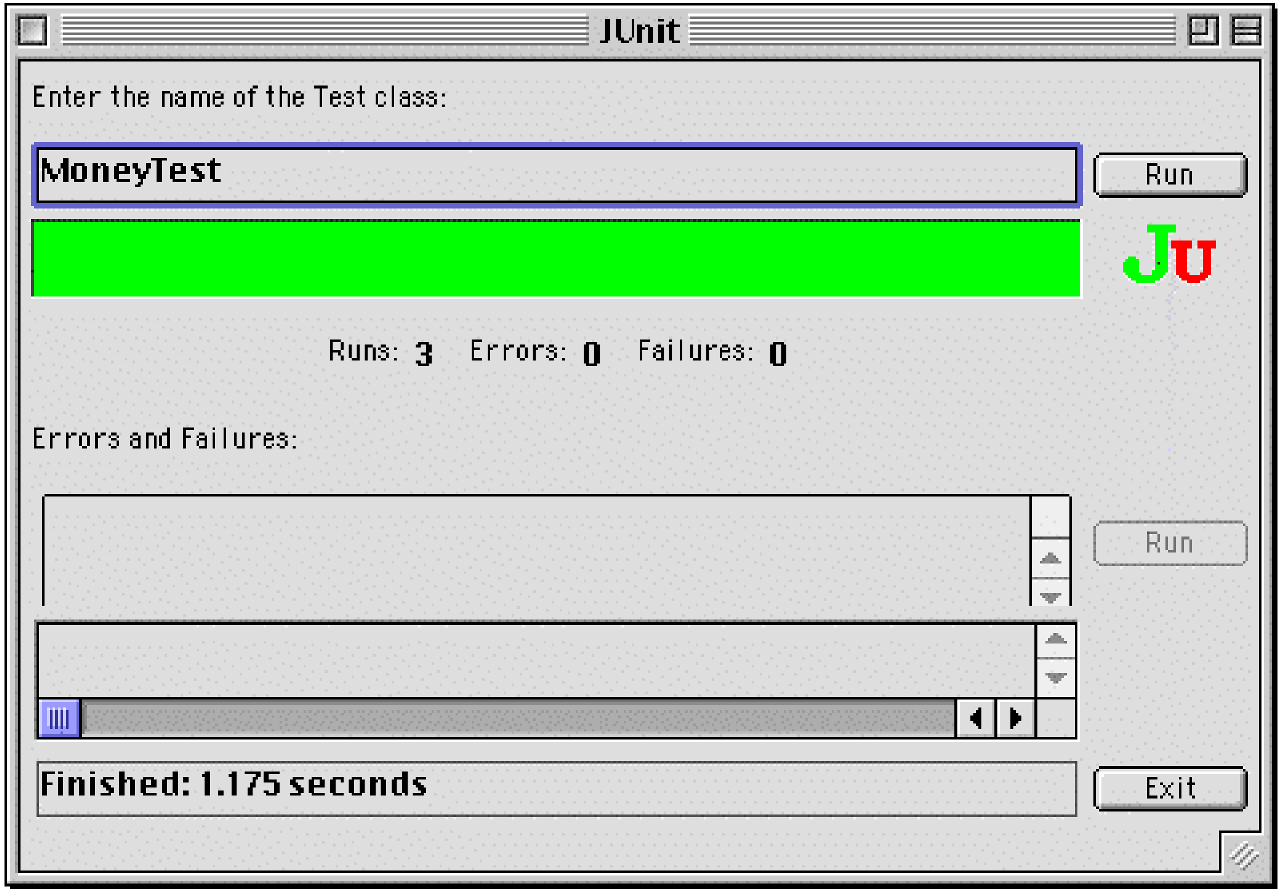
```
public void testBagEquals() {
  assert(!fMB1.equals(null) );
  assertEquals(fMB1, fMB1);
  assert(!fMB1.equals(f12CHF));
  assert(!f12CHF.equals(fMB1));
  assert(!fMB1.equals(fMB2));
}
```

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

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

Testing MoneyBags (III)

and run the tests.



Adding MoneyBags

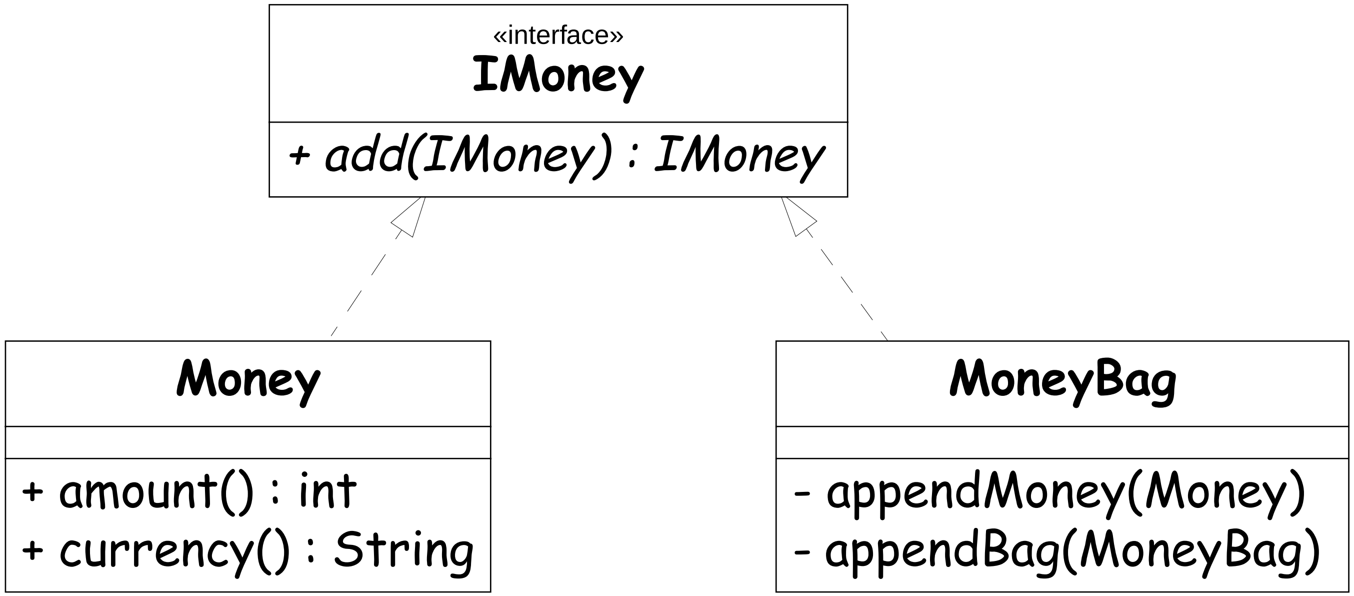
We would like to freely *add together arbitrary Monies and MoneyBags*, and be sure that *equals behave as equals:*

```
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?*

```
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 as a MoneyBag
  } ...
}
```

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

Double Dispatch (II)

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());
    else
      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 has to be:

«interface»
IMoney

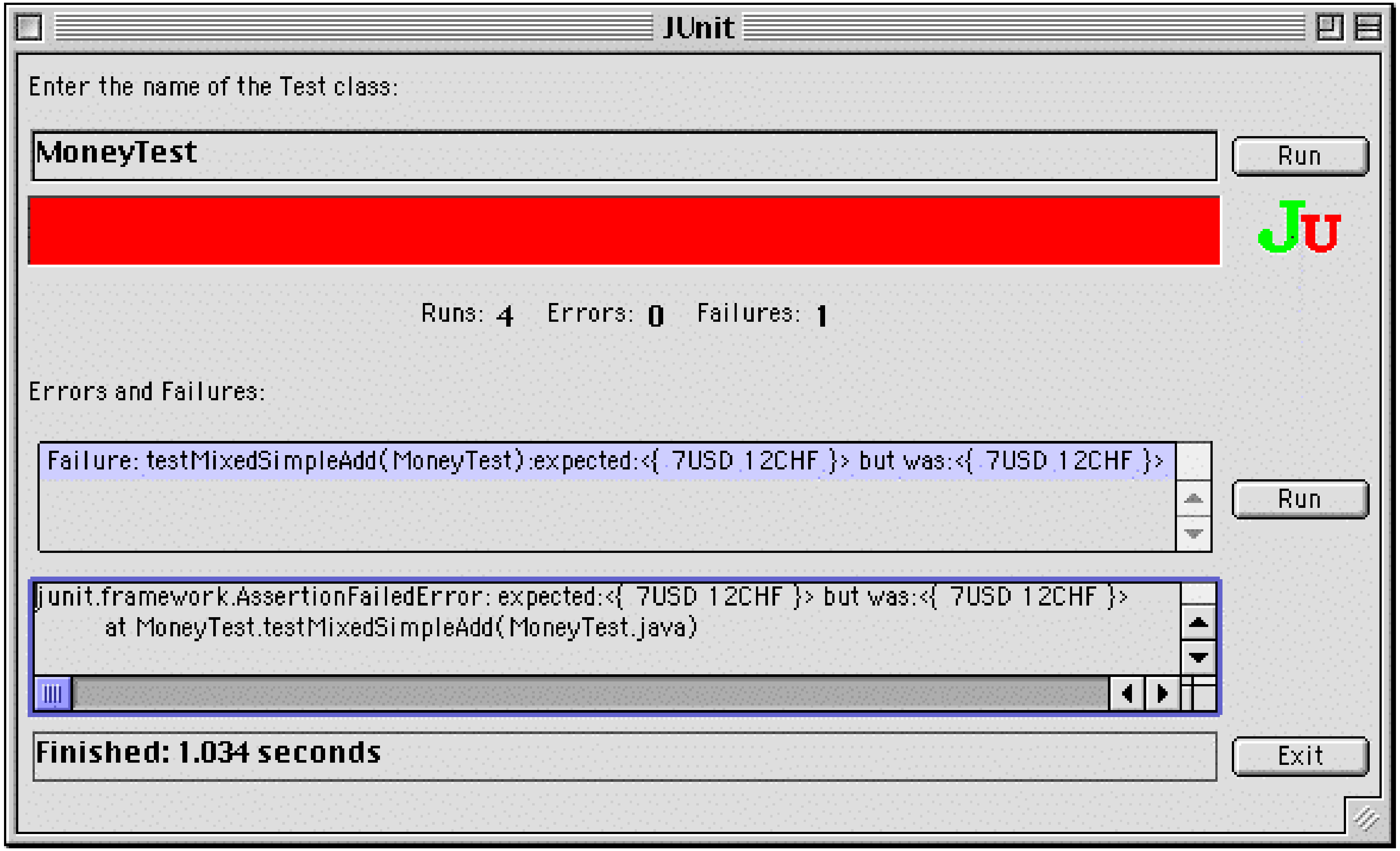
+ add(IMoney) : IMoney
+ addMoney(Money) : IMoney
+ addMoneyBag(MoneyBag) : IMoney

```
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 ...

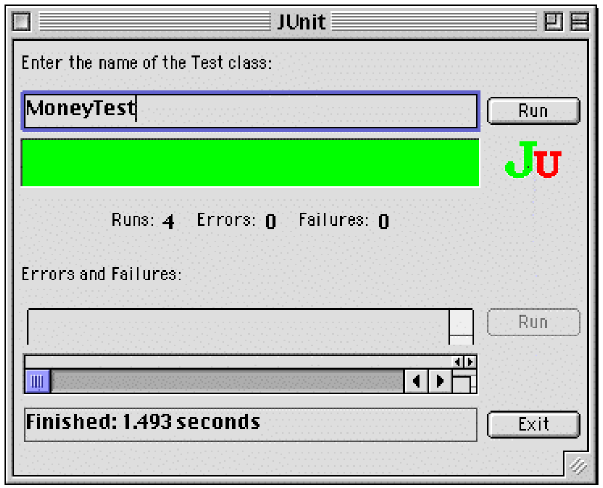


The fix ...

It seems we forgot to implement MoneyBag.equals()!

We fix it:

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



... test it, and continue developing.

What you should know!

- 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 these questions?

- *How does implementing toString() help in debugging?*
- *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

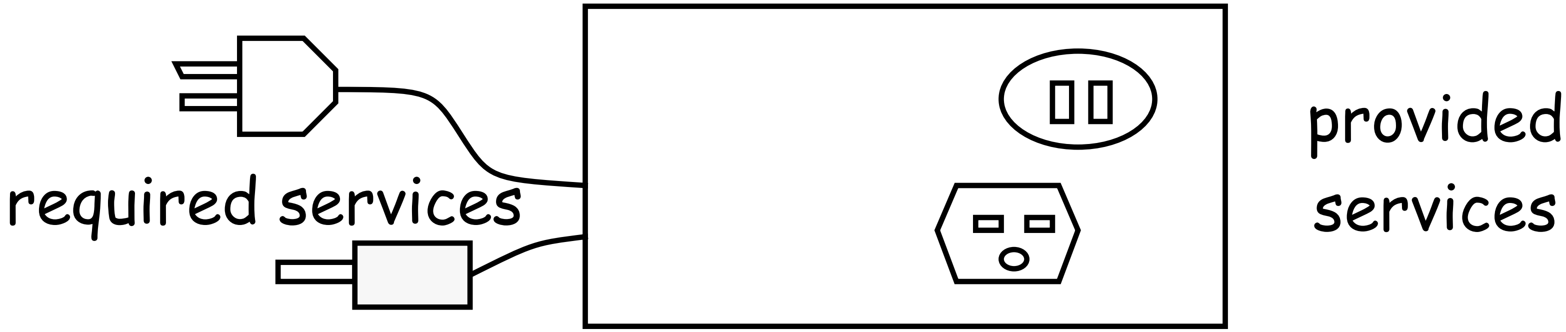
Source

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

Components

<u>*Components*</u> are *black-box* entities that:

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



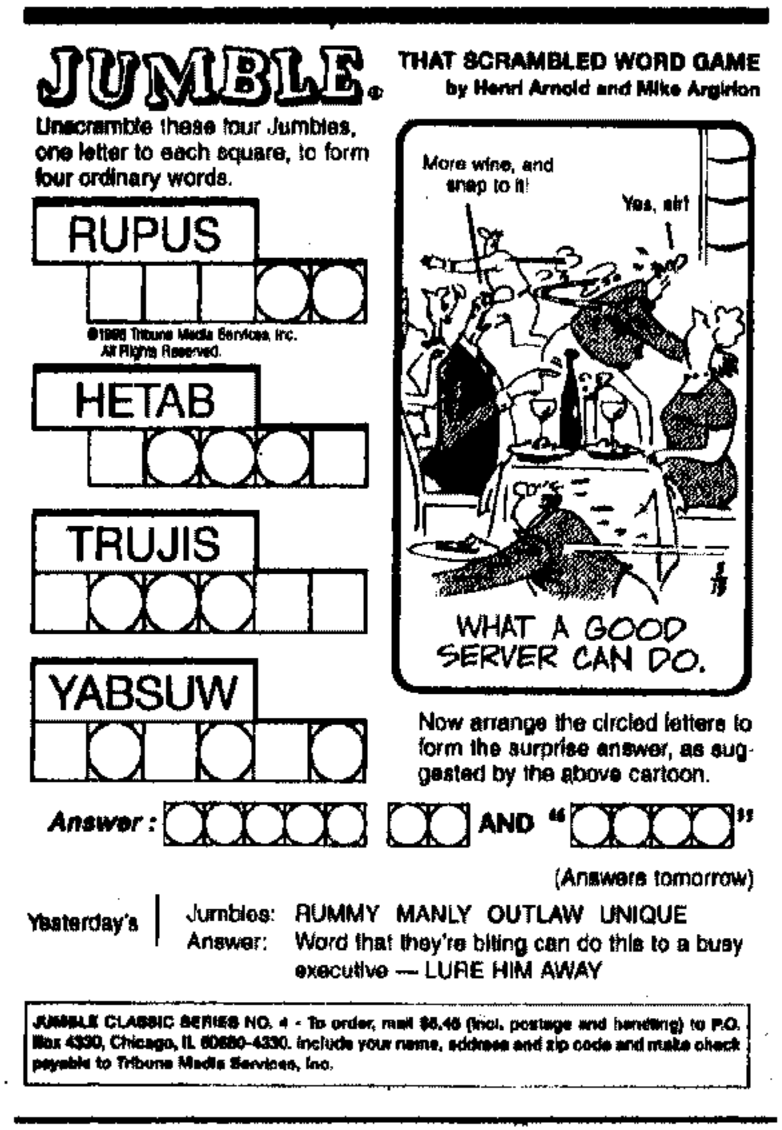
Components may be fine-grained (classes) or coarse-grained (applications).

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).



Naive Solution

Generate *all permutations* of the jumbled words:

rupus
urpus
uprus
purus
pruus
...

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

abacus
abalone
abase
...
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?

...

Rethinking the Jumble Problem ...

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

We can assign each word 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 simply 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:

Key	Word
aabcsu	abacus
aabelno	abalone
...	...
prsuu	*usurp*
...	...
chiruz	zurich
egotyz	zygote

2. Generate the key of a jumbled word:

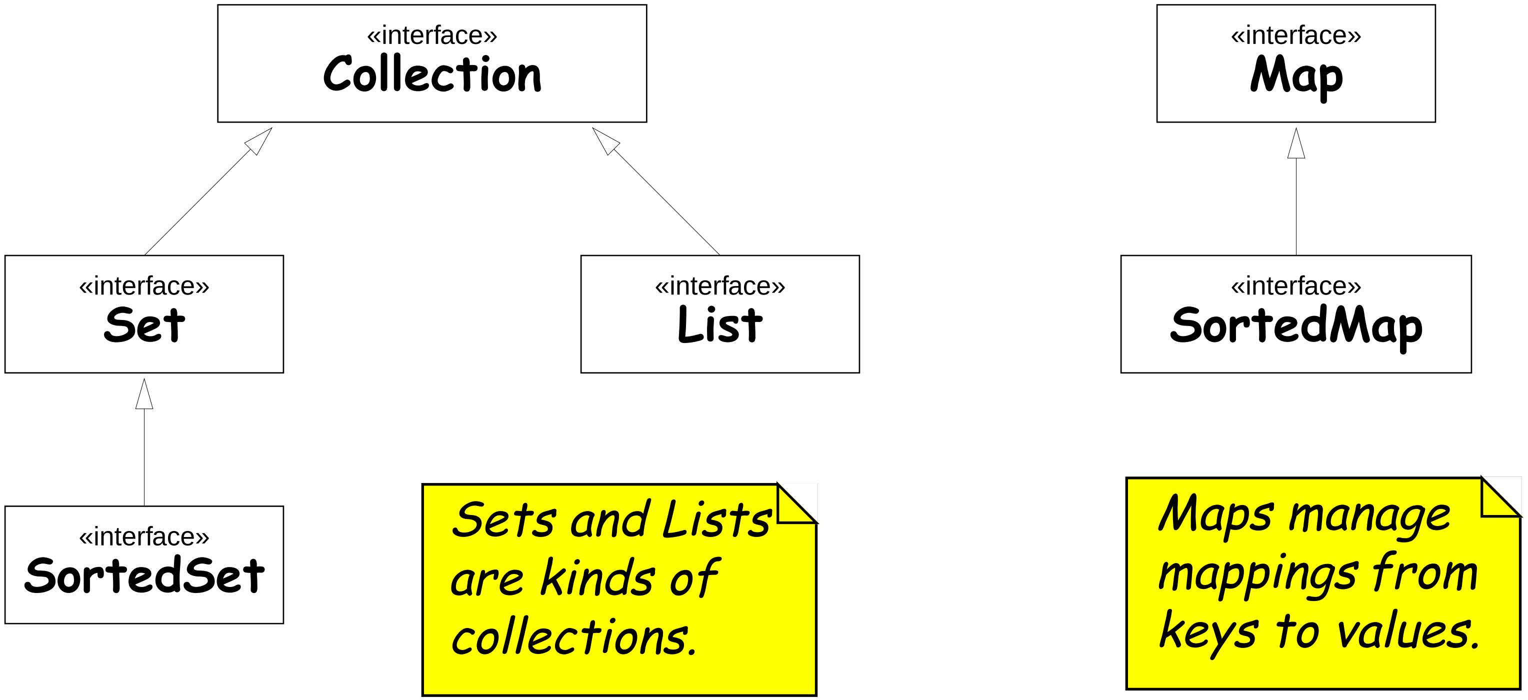
   key("rupus") = "prsuu"

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

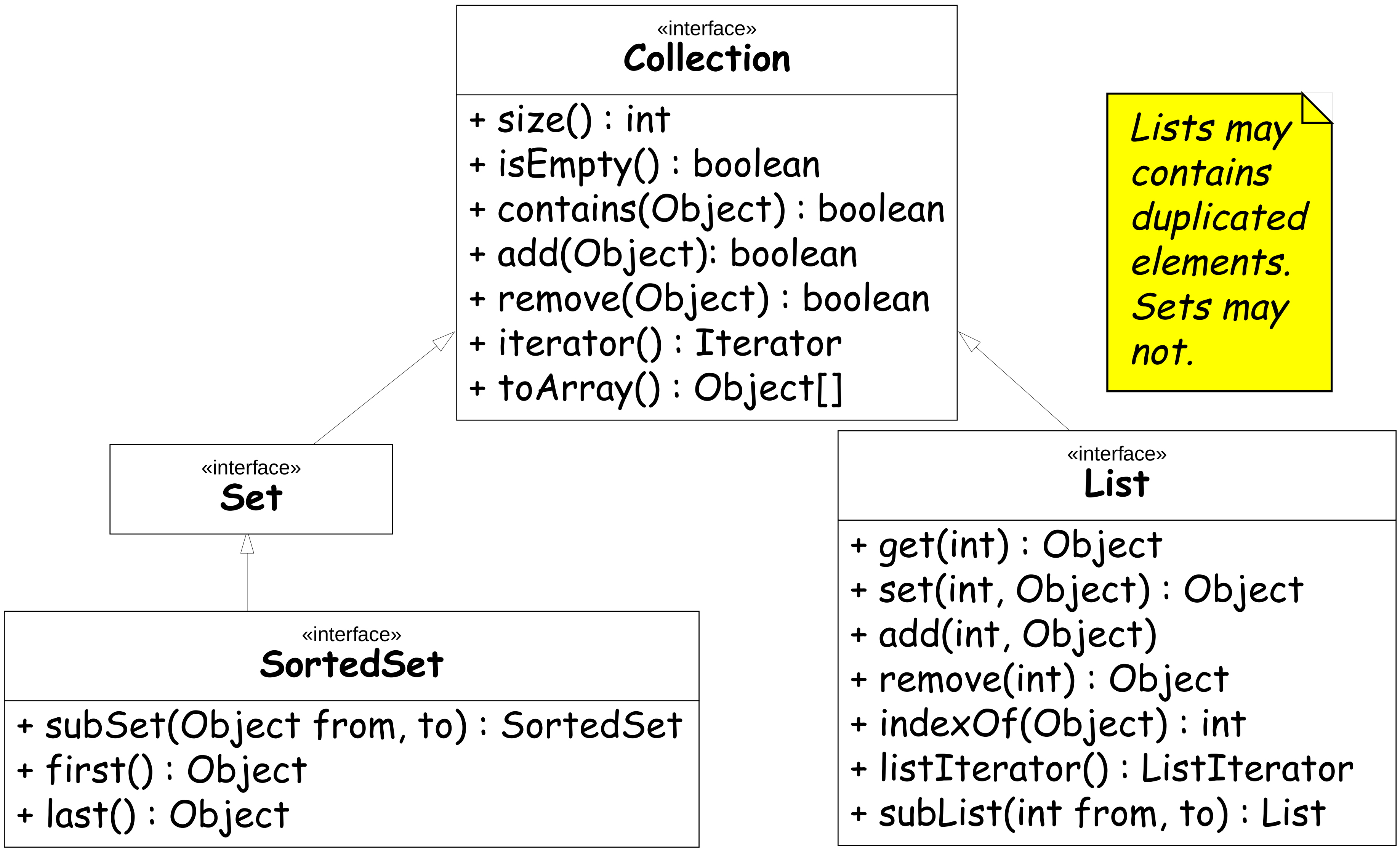
To implement a software solution, we need *associative arrays*, *lists*, *sort routines*, and possibly other components.

The Collections Framework

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

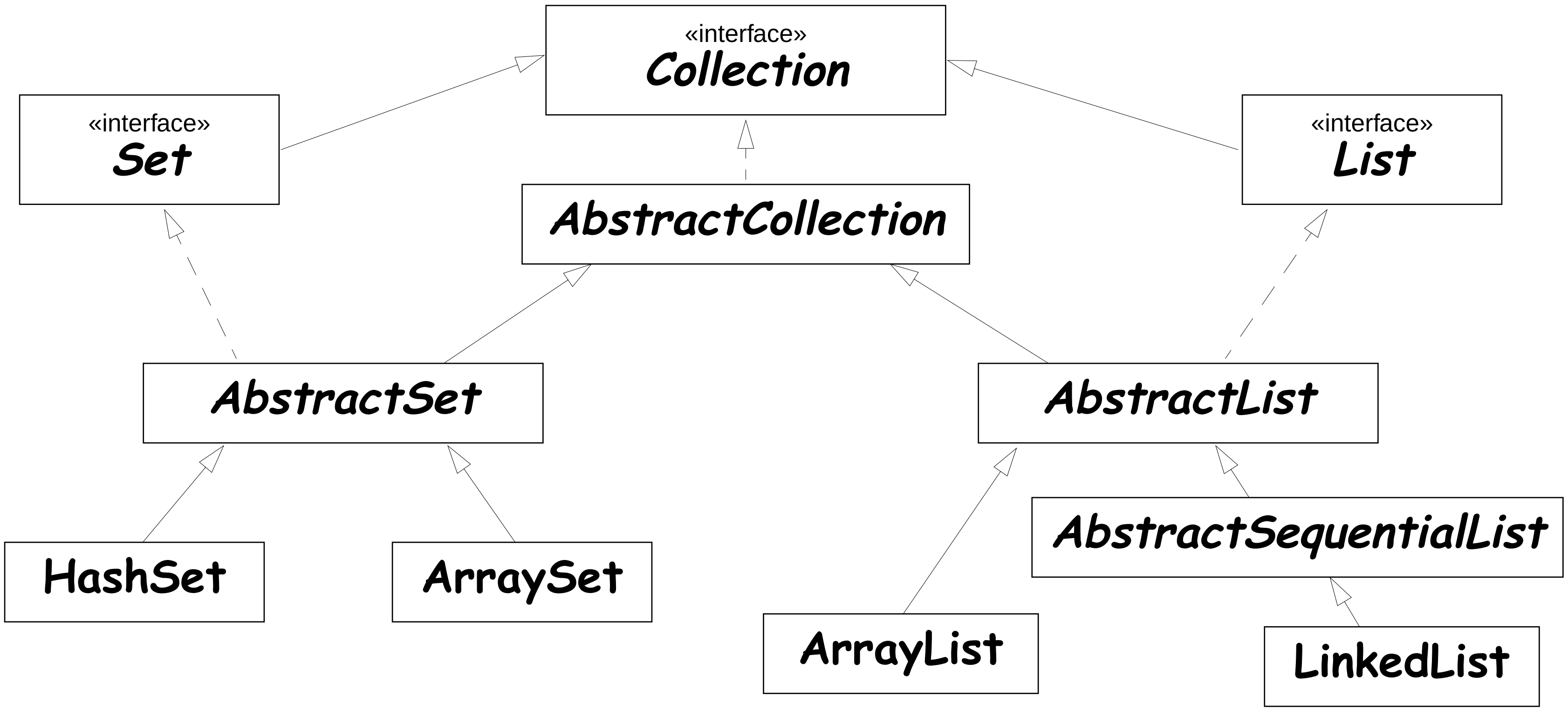


Collection Interfaces



Implementations

The framework provides at least two implementations of each interface.



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

Interface and Abstract Classes

Principles at play:

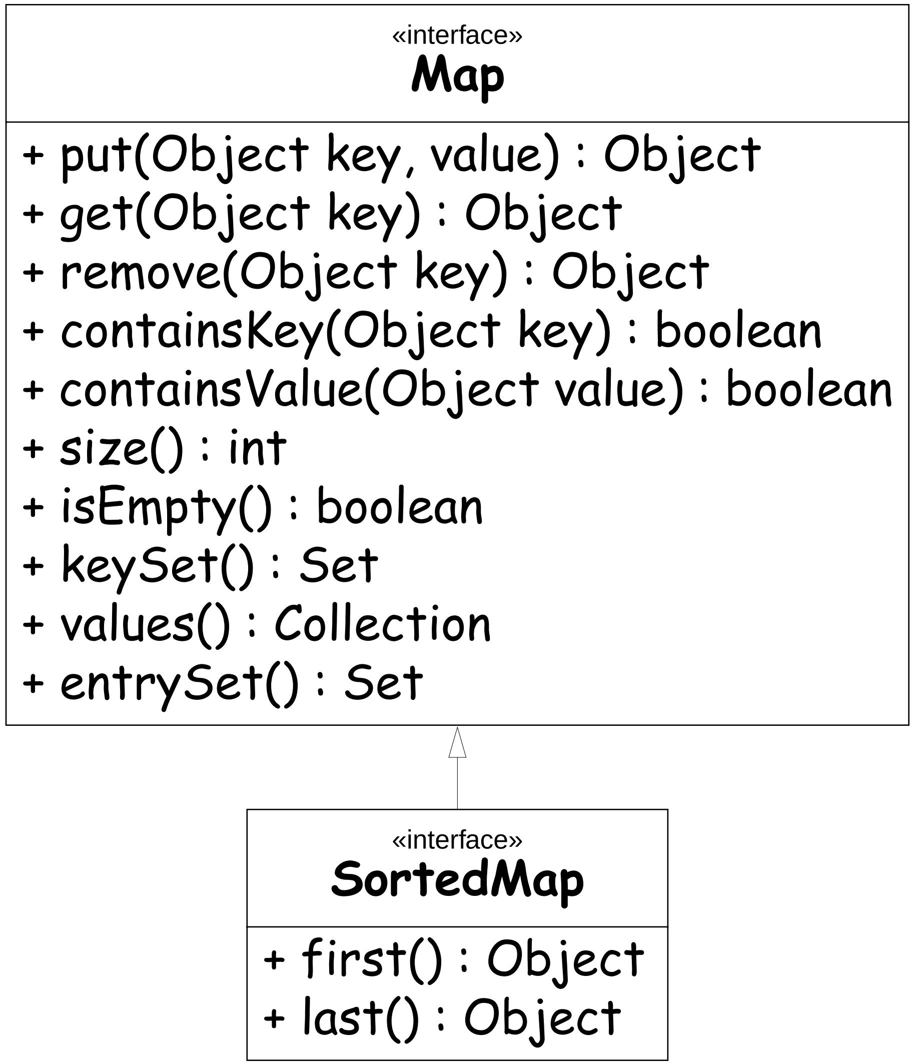
- Clients *depend only on interfaces*, not classes
- Classes may *implement multiple interfaces*
- Single inheritance doesn't prohibit *multiple subtyping*
- Abstract classes collect *common behaviour* shared by multiple subclasses but cannot be instantiated themselves, because they are incomplete

Maps

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

Map is implemented by HashMap and TreeMap.

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



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) { ... }
    Jumble wordMap = null;
    try { wordMap = new Jumble(args[0]); }
    catch (IOException err) {
      System.err.println("Can't load dictionary");
      return;
    }
    wordMap.inputLoop();
  }
...
```

Jumble constructor

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

```
private String wordFile_;

Jumble(String wordFile) throws IOException {
  super(); // NB: establish superclass invariant!
  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!)

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

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

✎ *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
... + <u>sort</u>(char[]) + <u>sort</u>(char[], int, int) + <u>sort</u>(double[]) + <u>sort</u>(double[], int, int) + <u>sort</u>(float[]) + <u>sort</u>(float[], int, int) + <u>sort</u>(int[]) + <u>sort</u>(int[], int, int) + <u>sort</u>(Object[]) + <u>sort</u>(Object[], Comparator) + <u>sort</u>(Object[], int, int) + <u>sort</u>(Object[], int, int, Comparator) ...

Sorting arrays of 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();
    }
  }
...
```

Loading the dictionary ...

... 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

Now the input loop is straightforward ...

```
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 ...”;
        } else {
          System.out.println(
              word + " unjumbles to: " + wordList);
        } ...
```

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!
```

Searching for anagrams

We would now like to know which word in the list has the largest number of *anagrams* — i.e., *what is the largest set of words with the same key.*

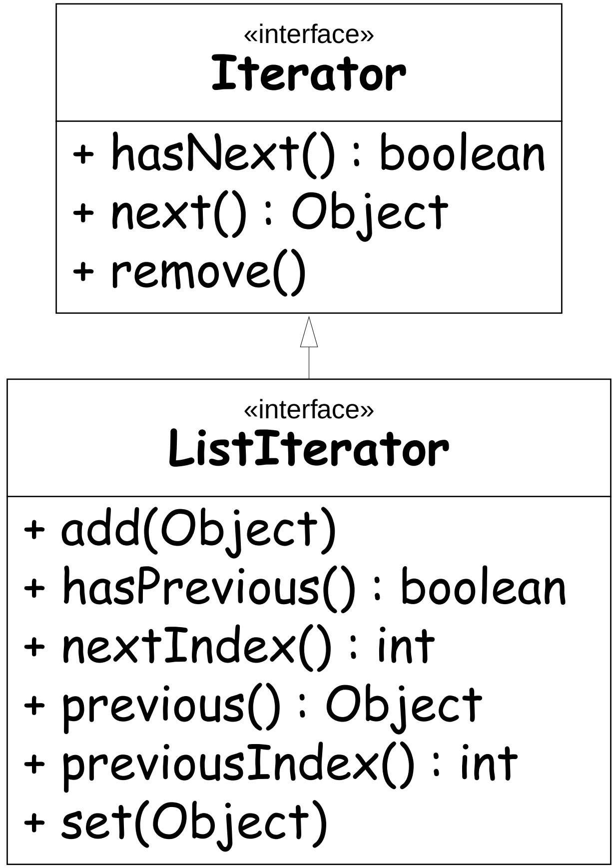
➤ How do you iterate through a Collection whose elements are unordered?

✔ *Use an iterator.*

Iterators

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

```
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;
}
```

Running Jumble.maxAnagrams

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).

What you should know!

- 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 these 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 Nutshell: 3d edition*, O'Reilly, 1999.
- 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

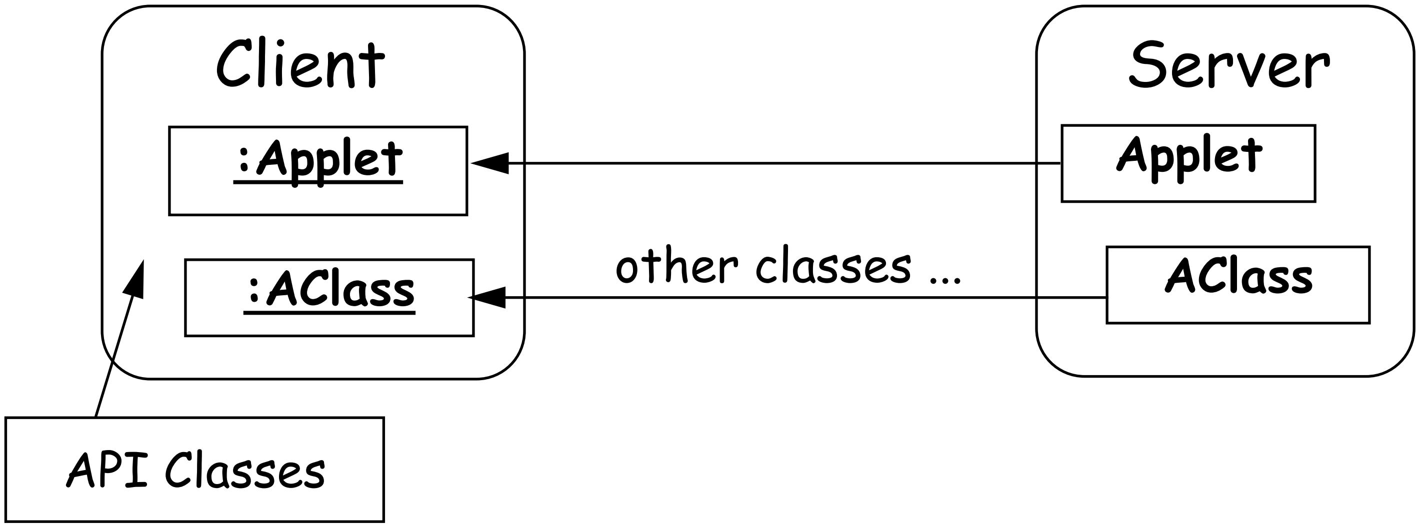
We would like to migrate it towards an interactive, network based game:

- 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.



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 );
  }
}
```

The Applet will be initialized and started by the client.

The Hello World Applet

```
<HTML>
<HEAD><TITLE>HelloApplet</TITLE></HEAD>
<BODY>
<APPLET
  CODEBASE = "."
  ARCHIVE  = "HelloApplet.jar"
  CODE     = "HelloApplet.class"
  NAME     = "HelloApplet"
  WIDTH    = 400
  HEIGHT   = 300
>
</APPLET>
</BODY>
</HTML>
```

Accessing the game as an Applet

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

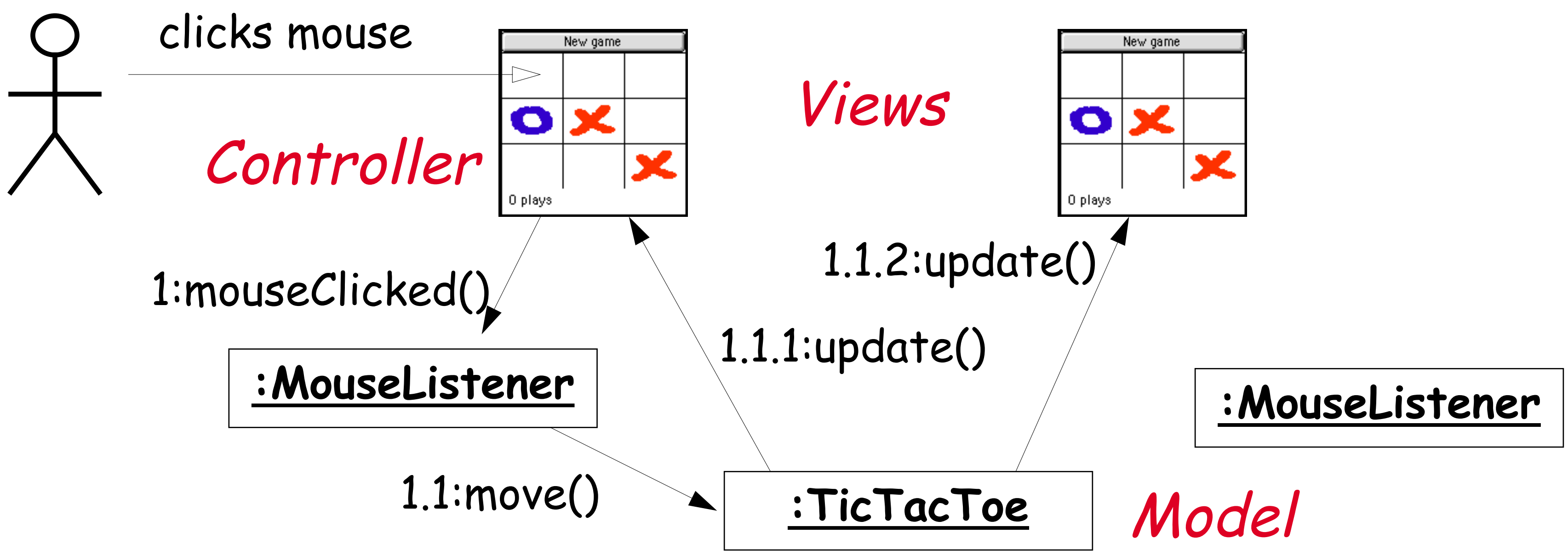
```
<title>GameApplet</title>
<applet
  codebase="AppletClasses"
  code="tictactoe.GameApplet.class"
  width=200
  height=200>
</applet>
```

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

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

Model-View-Controller

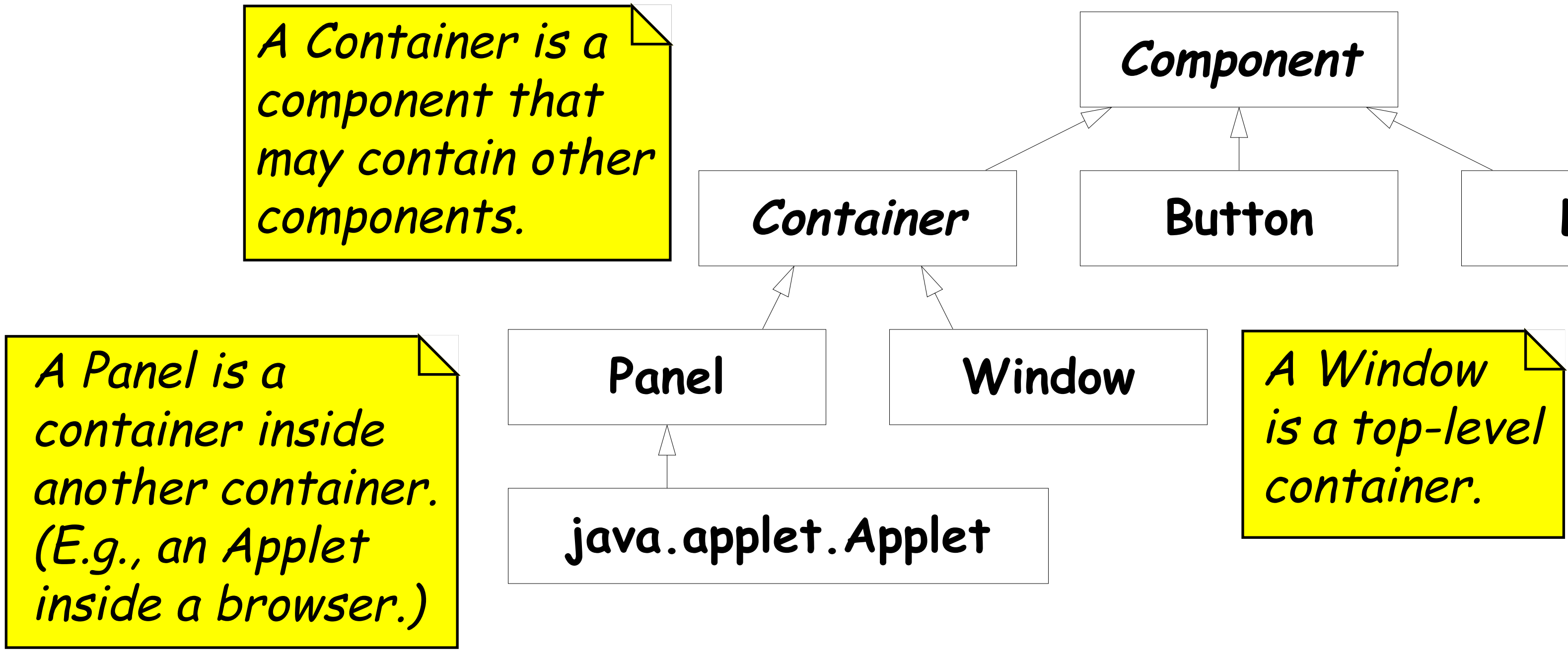
Version 1.6 of our game implements a *model* of the game, without a GUI. The GameApplet will implement a graphical *view* and a *controller* for GUI events.



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

AWT Components and Containers

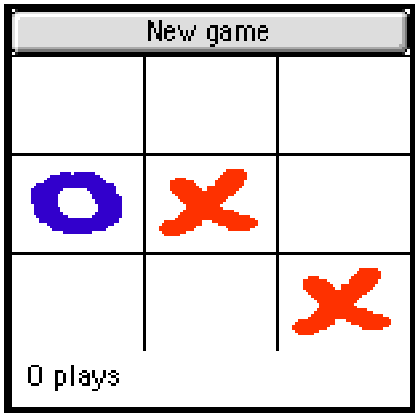
The java.awt package defines GUI *components*, *containers* and their *layout managers.*

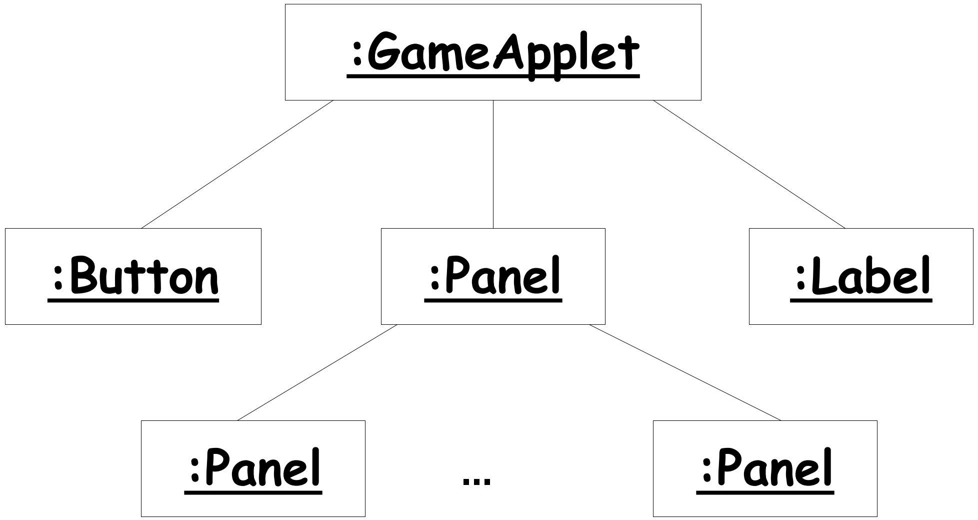


NB: There are also many graphics classes to define colours, fonts, images etc.

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

```
public void init() {
  game_ = makeGame();                    // instantiate game
  setLayout(new BorderLayout());  // initialize view
  setSize(MINSIZE*game_.cols(),
          MINSIZE*game_.rows());
  add("North", makeControls());
  add("Center", makeGrid());
  label_ = new Label();
  add("South", label_);
  game_.addObserver(this);           // connect to model
  showFeedBack(game_.currentPlayer().mark()
               + " plays");
}
```

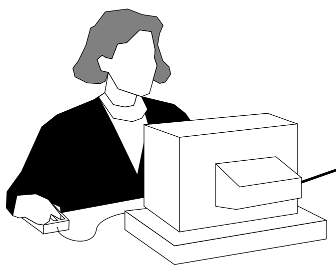
Helper methods

As usual, we introduce helper methods to hide the details of GUI construction ...

```
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 subscribed *Listener* objects

Callback methods

Hardware events ...

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

AWT Components *publish* events and (possibly multiple) Listeners *subscribe* interest in them.

Events and Listeners (II)

Every AWT component publishes a variety of different events (see java.awt.event) with associated Listener interfaces).

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()
...			

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();    // NB: has access to methods
    }               // of enclosing class!
  });
  return again;
}
```

We instantiate an *anonymous inner class* to avoid defining a named subclass of ActionListener.

Listening for mouse clicks

We also *attach a MouseListener* to each Place on the board.

```
private Component makeGrid() { ...
  Panel grid = new Panel();
  grid.setLayout(new GridLayout(rows, cols));
  place_s = new Place[cols][rows];
  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));
      ...
  return grid;
}
```

The PlaceListener

MouseAdapter is a *convenience class* that defines *empty* MouseListener methods (!)

```
public class PlaceListener extends MouseAdapter {
  private final Place place_;
  private final GameApplet applet_;
  public PlaceListener(...) {
    place_ = place;
    applet_ = applet;
  }
...
```

The PlaceListener ...

We only have to define the mouseClicked() method:

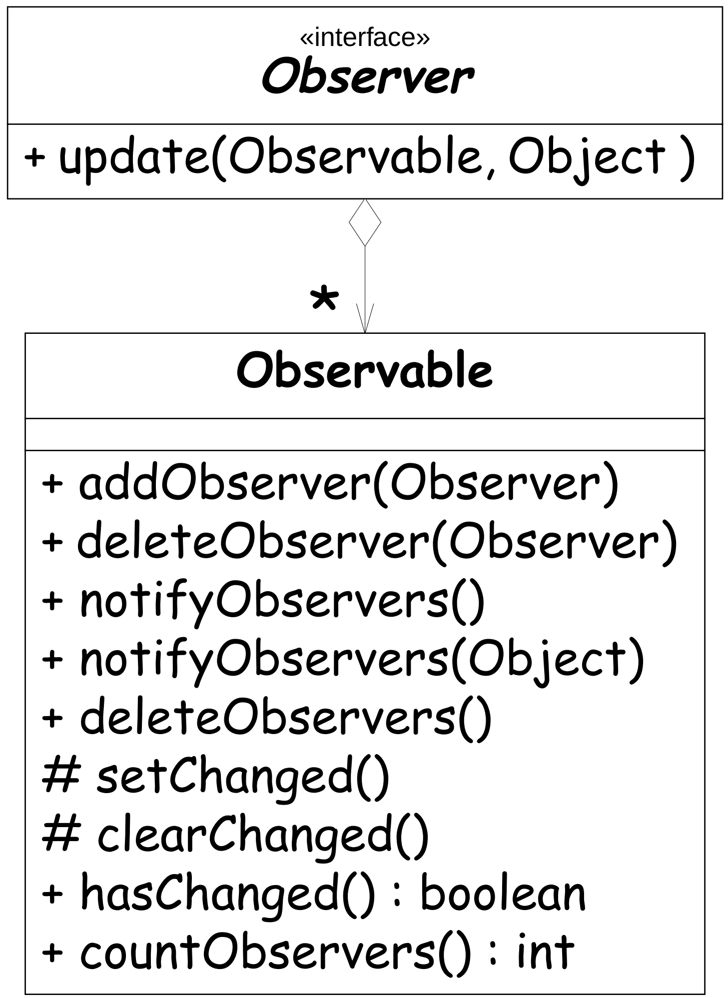
```
public void mouseClicked(MouseEvent e){
   ...
   if (game.notOver()) {
      try {
         ((AppletPlayer) game.currentPlayer()).move(col,row);
         applet_.showFeedBack(game.currentPlayer().mark() + " plays");
      } catch (AssertionException err) {
         applet_.showFeedBack("Invalid 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

In our case, the GameApplet represents a *View*, so plays the role of an *Observer:*

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

Observing the BoardGame ...

The BoardGame represents the *Model*, so plays the role of an *Observable:*

```
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));
  }
}
```

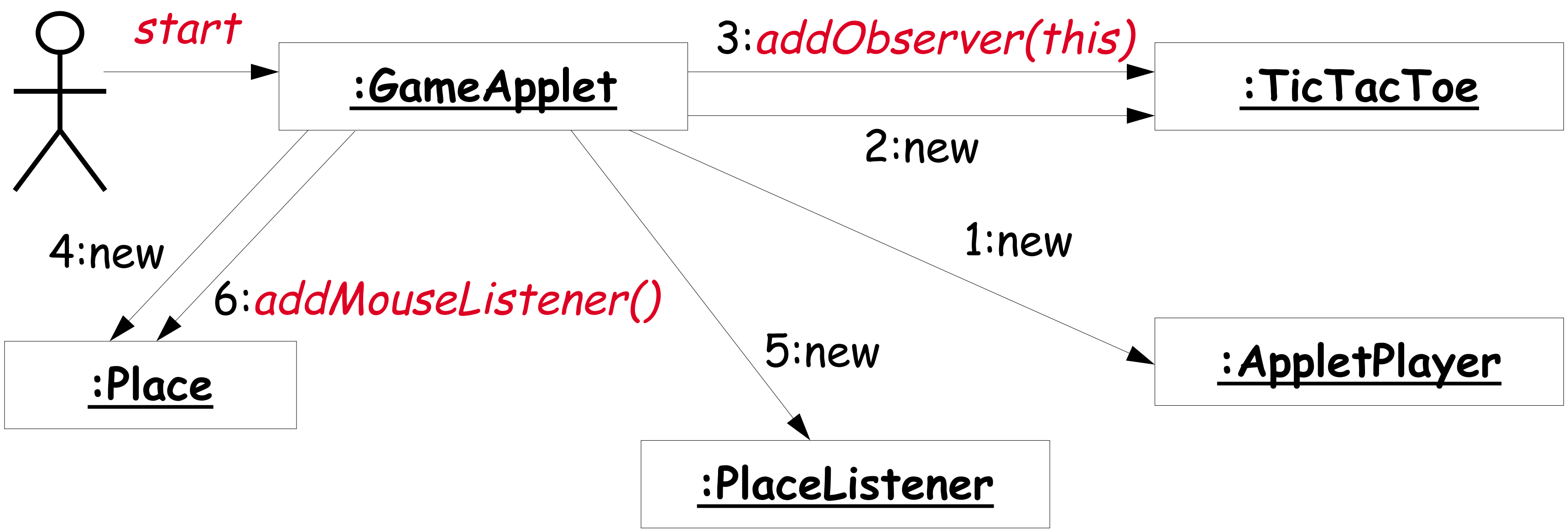
Communicating changes

A Move instance bundles together information about a change of state in a BoardGame:

```
public class Move {
  public final int col, row; // NB: public, but final
  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

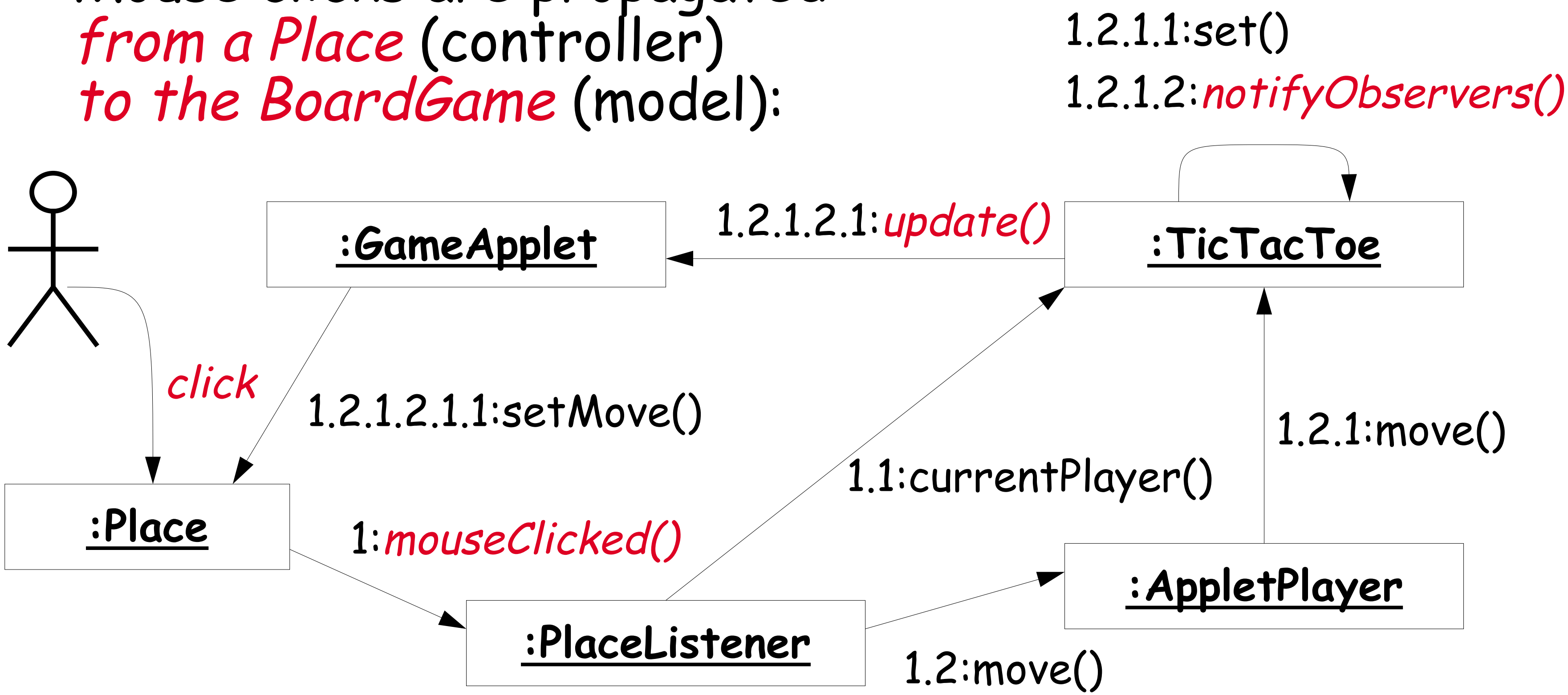
When the GameApplet is loaded, its init() method is called, causing the *model*, *view* and *controller* components to be *instantiated.*



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

...

Refactoring the BoardGame ...

- 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.

What you should know!

- 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 these 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

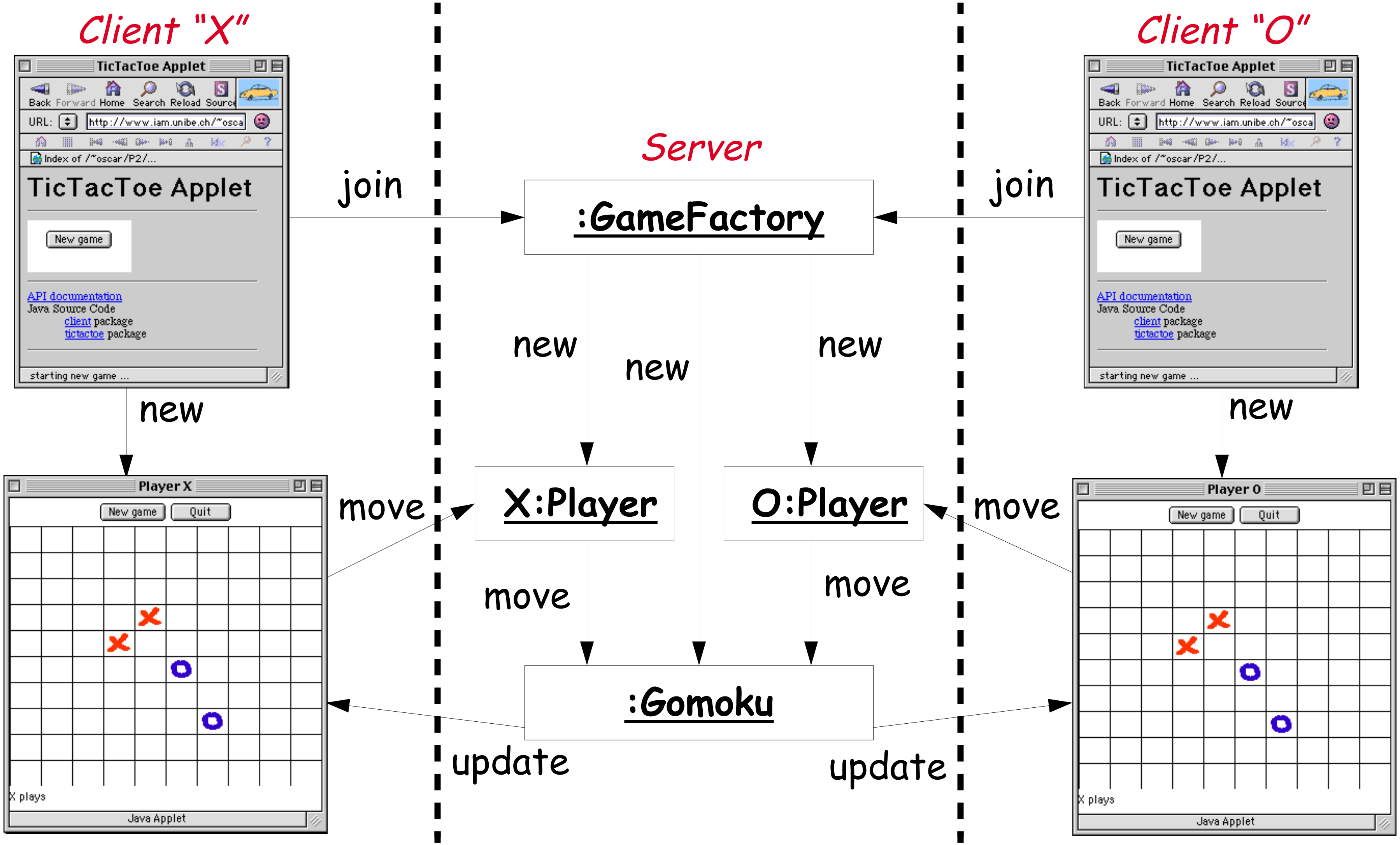
A Networked TicTacToe?

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"
Client "O"
Server
:GameFactory
X:Player
O:Player
:Gomoku
join
join
new
new
new
new
new
move
move
move
move
update
update
TicTacToe Applet
New game
API documentation
Java Source Code
client package
tictactoe package
starting new game ...
Player X
Player O
Quit
X plays
Java Applet

The problem

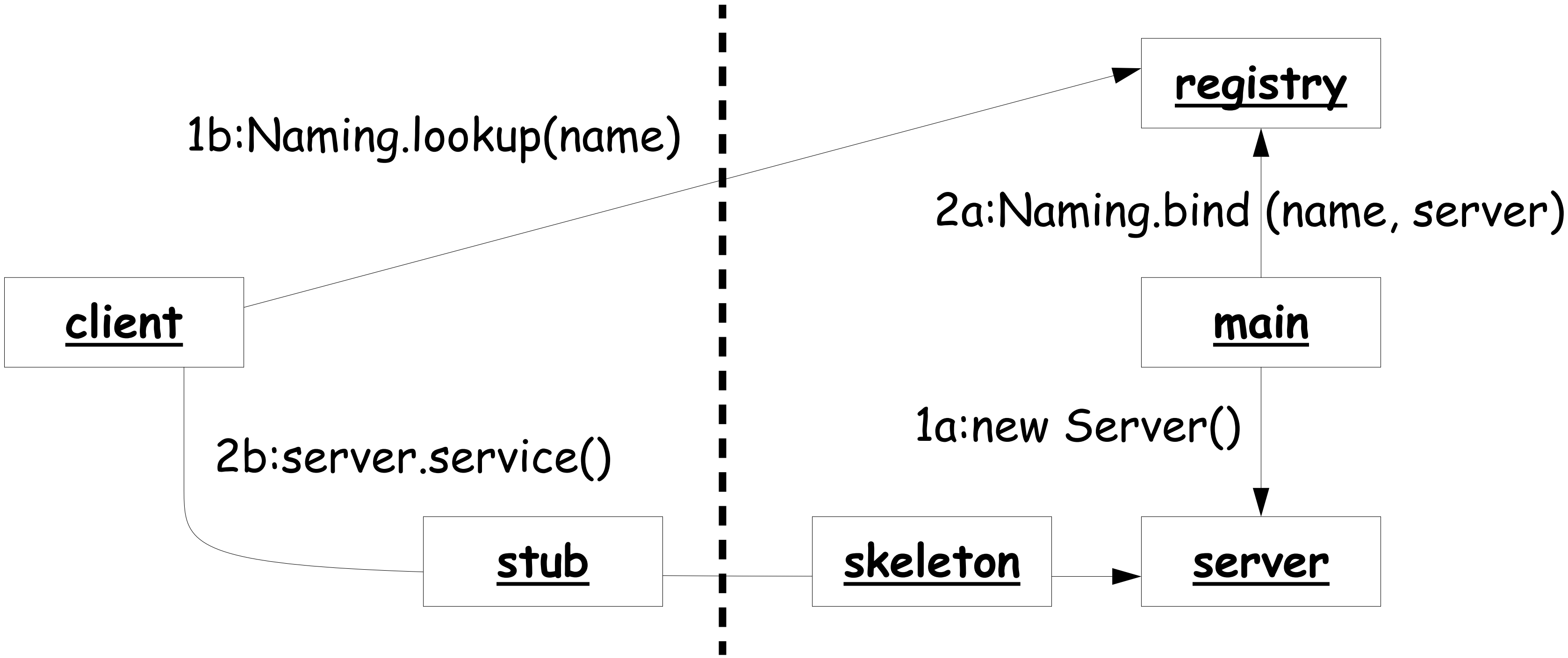
Unfortunately *Applets alone are not enough* to implement this scenario!

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 (*stub*) that acts as a *proxy* for the remote server object (represented by a *skeleton*).

Why do we need RMI?

RMI

- hides complexity of network protocols
- offers a standard rmiregistry implementation
- automates marshalling and unmarshalling of objects
- automates generation of stubs and skeletons

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*

...

Developing an RMI application ...

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*

BoardGame client/server interfaces

We split the game into three packages:

- **client** — contains the GUI components (view), 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 interface is 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

RemoteGame *exports only what is needed* by the client:

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

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) { ... }
  public String toString() { ... }
}
```

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();
  }
...
```

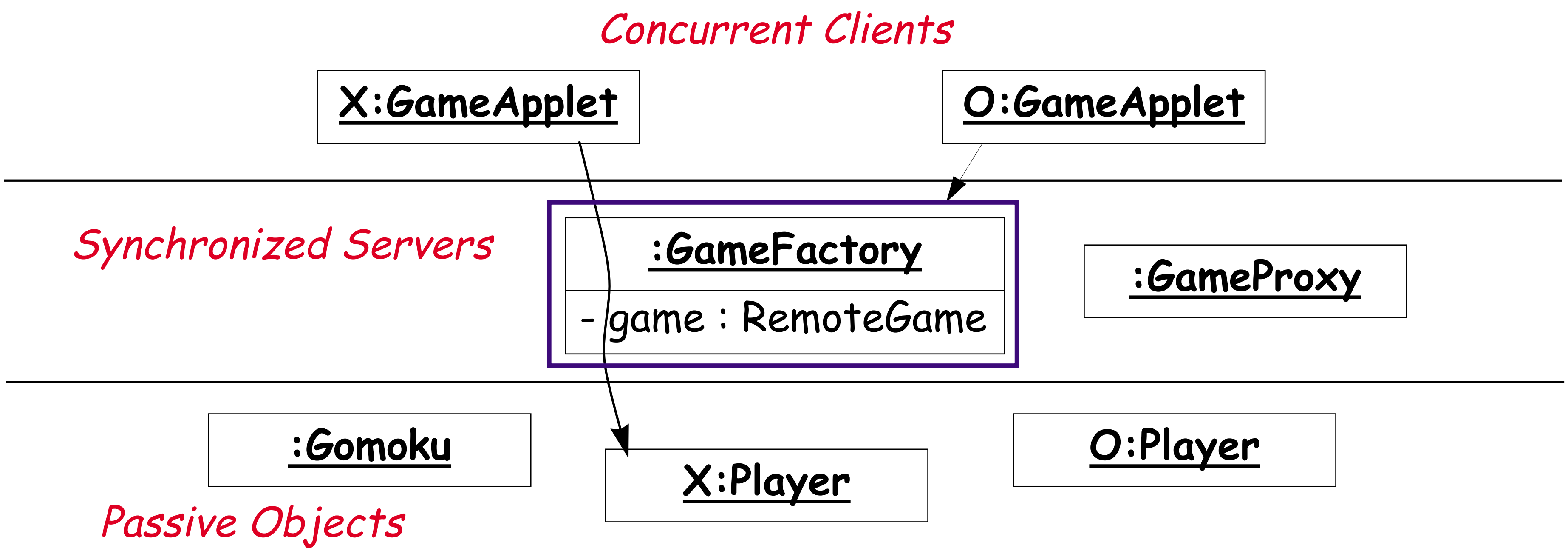
NB: All constructors for Remote objects must throw RemoteException!

Implementing Remote objects ...

```
...
  public synchronized RemoteGame joinGame()
                            throws RemoteException
  {
    RemoteGame game = game_;
    if (game == null) { // first player => new game
      game = new GameProxy(new Gomoku( ...));
      game_ = game;
    } else { game_ = null; }
    // second player => join existing game
    return game;
  }
}
```

A simple view of synchronization

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



➤ <u>How can servers protect their state from concurrent requests?</u>

✔ *Declare their public methods as synchronized.*

Registering a remote object

The server must be started by an ordinary main() method:

```
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(
                                new RMISecurityManager());
        System.out.println("Set new Security manager");
    }
...
```

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

Registering a remote object ...

The main() method must *instantiate* a GameFactory and *register* it with a running RMI registry.

```
...
    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., www.iam.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

We must prevent the server from being blocked by a call to the remote client.

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

```
class WrappedObserver implements Observer {
  private RemoteObserver remote_;

  WrappedObserver(RemoteObserver ro) {
    remote_ = ro;
  }

...
```

Using Threads to protect the server ...

```
public void update(Observable o, Object arg) {
  final Move move = (Move) arg; // for inner class
  Thread doUpdate = new Thread() {
    public void run() {
      try {
        remote_.update(move);
      } catch(RemoteException err) { }
    }
  };
  doUpdate.start() ;     // start the Thread
}                        // and ignore results
}
```

Even if the Thread blocks, the server can continue ...

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)
- 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*)
  - view now uses Move and RemoteGame (not 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.

Generating Stubs and Skeletons

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 start the RMI registry on the host (www.iam.unibe.ch):

```
rmiregistry 2001 &
```

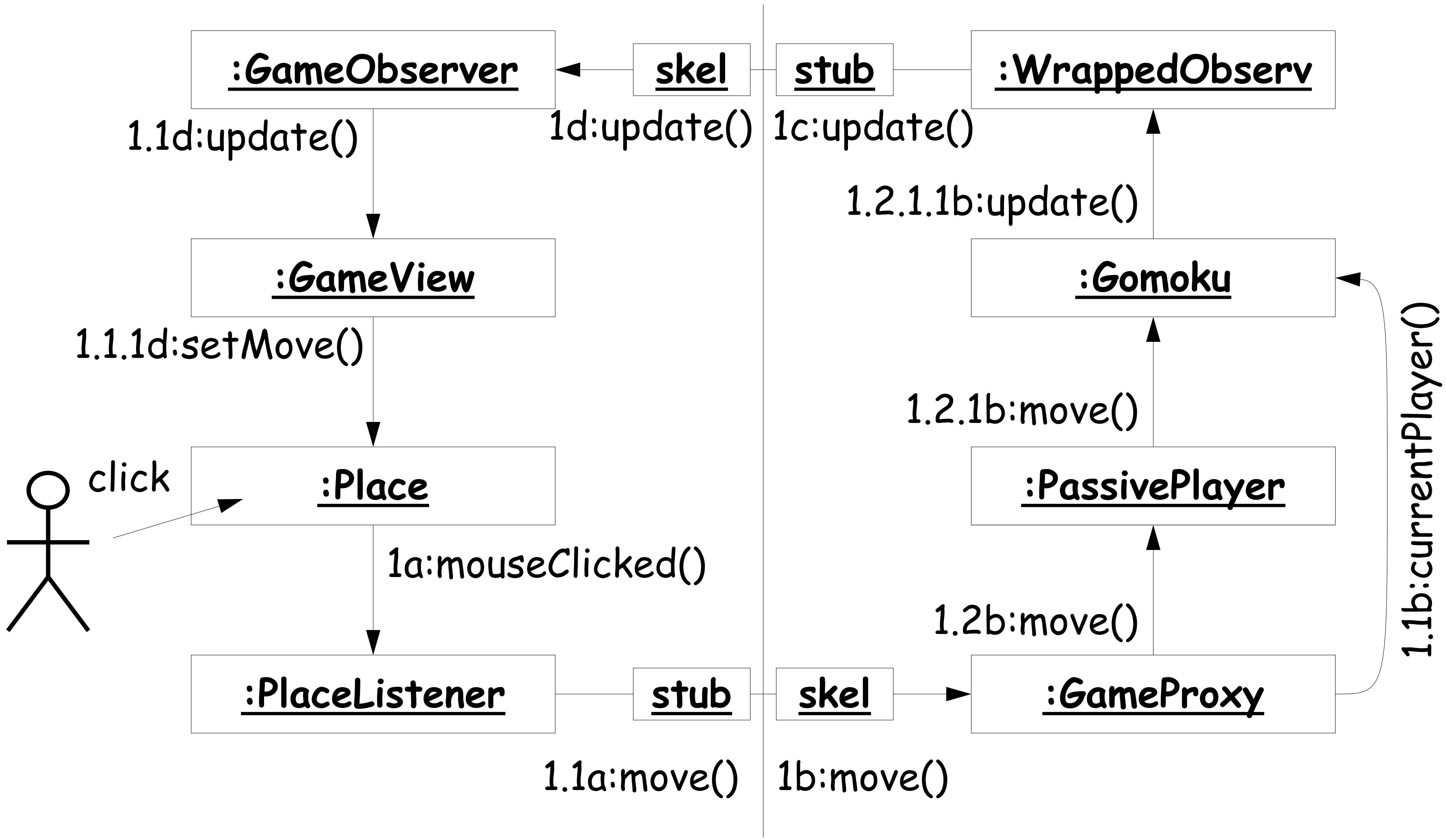
We start and register the servers:

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



Caveat!

This only works with JDK 1.1:

- ❑ Most web browsers are not Java 1.2 enabled
- ❑ Applets can only connect to the host of their codebase
- ❑ Security is more complex in Java 1.2
  - ☞ clients must specify a *policy* file

Web browsers, Applets, RMI and Java security don't mix well.

If you plan to use RMI and Java 2, stay away from applets!

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

What you should know!

- 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 these 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?*
- *What exactly happens when you send an object over the net via RMI?*

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 (high *coupling*)
  - ☞ redistribute responsibilities

Many idioms and patterns can help to improve your design ...

What are Idioms and Patterns?

Idioms	Idioms are common programming *techniques* and *conventions*. They are often language-specific.
Patterns	Patterns document *common solutions* to *design problems*. They are language-independent.
Libraries	Libraries are *collections of functions*, procedures or other software components 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 common idioms and patterns.

Delegation

➤ <u>How can an object share behaviour without inheritance?</u>

✔ *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.

Delegation

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 {
  ...
  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

➤ <u>How do you extend behaviour inherited from a superclass?</u>

✔ *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.

Super

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 — 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

➤ <u>How do you keep a client of a service independent of classes that provide the service?</u>

✔ *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.

Interface

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.

Adapter

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();
    }
  });
  return again;
}
```

Proxy

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

✔ *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.

Proxy

Example

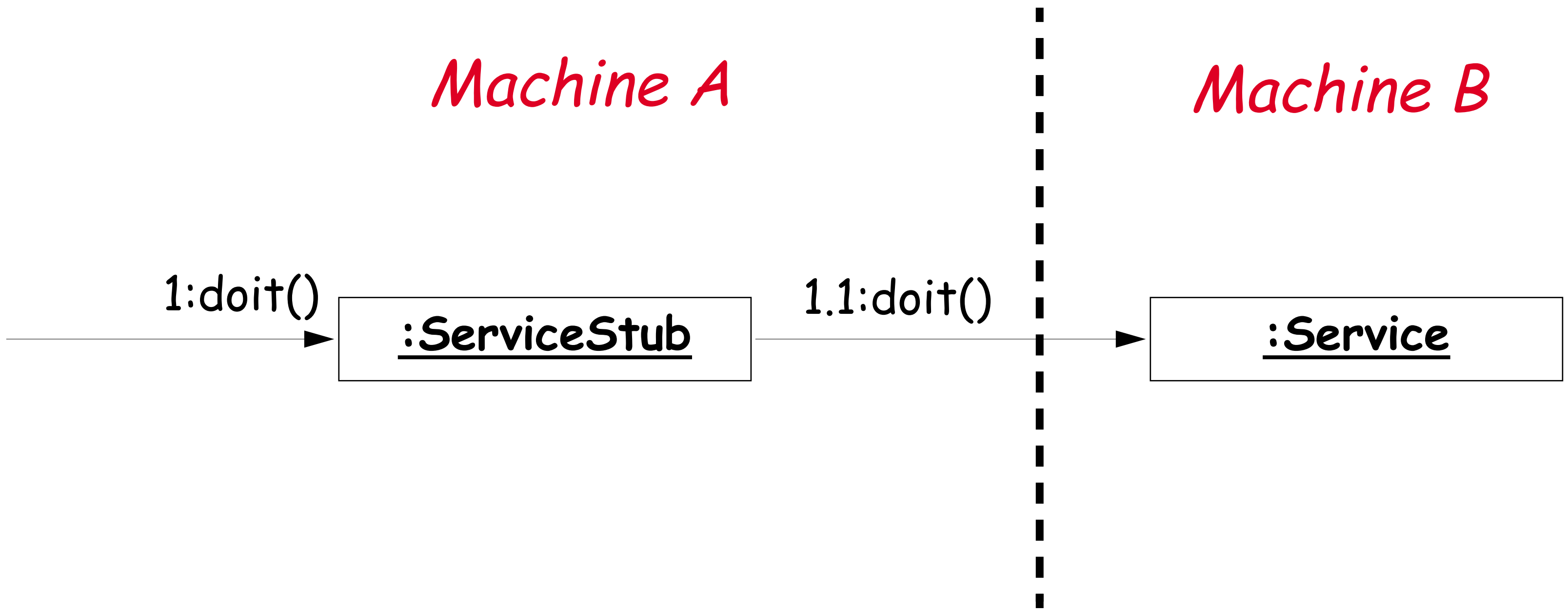
- A Java “stub” for a remote object accessed by Remote Method Invocation (RMI).

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 *delegates* the rest to:

- *hook methods* that subclasses *may extend*, and
- *abstract methods* that subclasses *must implement*.

Template Method

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(); }
    finally { tearDown(); }
  }
  protected void setUp() { }  // empty by default
  protected void tearDown() { }
  protected void runTest() throws Throwable { ... }
}
```

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.

Composite

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.

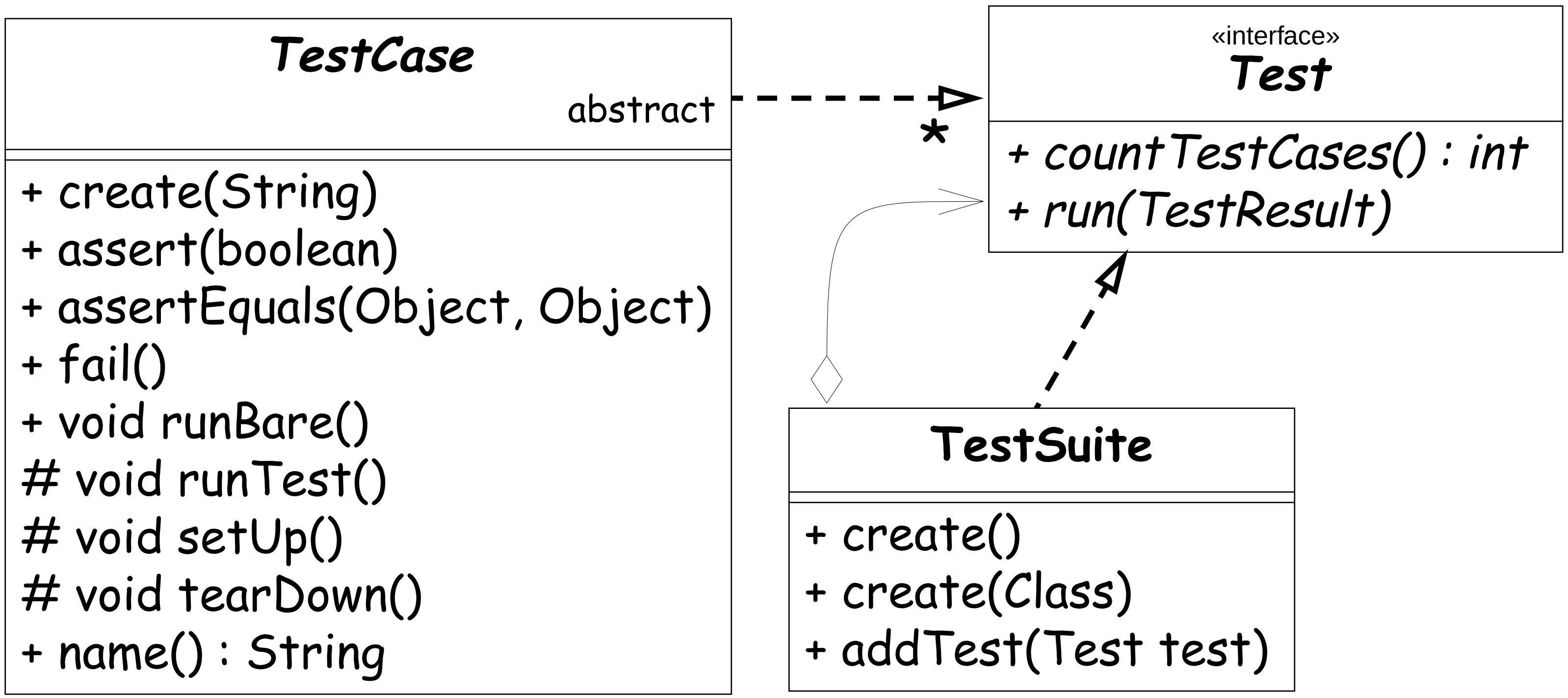
Consequences

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

➤ <u>How can an object inform arbitrary clients when it changes state?</u>

✔ *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.

Observer

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

Doug Schmidt, CACM Oct 1995

What you should know!

- 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 these 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 ...
- A few Java puzzles ...

Sources

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

Round-off errors

What does this print?

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

println(100*(f-g));
```

== versus equals() (1)

When are two Strings equal?

```
String s1 = new String("This is a string");
String s2 = new String("This is a string");
test("String==", s1 == s2);
test("String.equals", s1.equals(s2));
```

```
static void test(String name, boolean bool) {
  println(name + ": " + (bool?"true":"false"));
}
```

== versus equals() (2)

When are two Objects equal?

```
Object x = new Object();
Object y = new Object();
test("object==", x == y);
test("object.equals", x.equals(y));
```

== versus equals() (3)

When are two Strings equal?

```
String s3 = "This is a string";
String s4 = "This is a string";
test("String==", s3 == s4);
test("String.equals", s3.equals(s4));
```

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))
  println("see you later");
else
  println("see you now");
```

The dangling else problem.

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

What is printed when we run these checks?

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

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?

```
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;
}
```

Avoiding Off-by-1 errors

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!

For which values does this function work correctly?

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

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 a clone* instead)

Bugs are always matter of invalid assumptions not holding

Puzzle 1

Are private methods inherited?

```
class A {
  public void m() { this.p(); }
  private void p() { println("A.p()"); }
}
class B extends A {
  private void p() { println("B.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 are overloaded method calls resolved?

```
class A { }
class B extends A { }
void m(A a1, A a2) { println("m(A,A)"); };
void m(A a1, B b1) { println("m(A,B)"); };
void m(B b1, A a1) { println("m(B,A)"); };
void m(B b1, B b2) { println("m(B,B)"); };
B b = new B(); A a = b;
```

Which is considered: the *static* or *dynamic* argument type?

```
m(a, a);
m(a, b);
m(b, a);
m(b, b);
```

Puzzle 2 (part II)

What happens if we comment out:

- m(A,A)?
- m(B,B)?
- m(A,B)?

Will the examples still compile?
If so, which methods are called?

Puzzle 3

How do static and dynamic types interact?

```
class A {
  void m(A a) { println("A.m(A)"); }
}
class B extends A {
  void m(B b) { println("B.m(B)"); }
}
B b = new B(); A a = b;
```

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

```
a.m(a);
a.m(b);
b.m(a);
b.m(b);
```

Puzzle 4 (part I)

How do default values and constructors interact?

```
class C {
  int i = 100, j = 100, k = init(), l = 0;
  C() { i = 0; k = 0; }
  int init() { j = 0; l = 100; return 100; }
}
```

What gets printed? 0 or 100?

```
C c = new C();
println("C.i = " + c.i);
println("C.j = " + c.j);
println("C.k = " + c.k);
println("C.l = " + c.l);
```

Puzzle 4 (part II)

```
abstract class A {
  int j = 100;
  A() { init(100); j = 200; }
  abstract void init(int value);
}
class B extends A {
  int i = 0, j = 0;
  B() { super(); }
  void init(int value) { i = value; }
}
```

What gets printed? 0, 100 or 200?

```
B b = new B();
println("B.i = " + b.i);
println("B.j = " + b.j);
```

Puzzle 5

Does try or finally return?

```
class A {
  int m() {
    try { return 1; }
    catch (Exception err) { return 2; }
    finally { return 3; }
  }
}
```

Prints 1, 2, or 3?

```
A a = new A();
println(a.m());
```

What you should know!

- *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 these 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?*