Principles of Software Construction: Objects, Design, and Concurrency (Part 1: Designing Classes)

Design for Change (class level)

Jonathan Aldrich Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Tradeoffs?

```
void sort(int[] list, boolean inOrder) {
   …
  boolean mustswap;
  if (inOrder) {
    mustswap = list[i] < list[j];
  } else {
    mustswap = list[i] > list[j];
  }
  …
}
```

```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int i, int j) { return i<j; }}

class DownComparator implements Comparator {
  boolean compare(int i, int j) { return i>j; }}
```

Source: BC Forestry website

Case Study: Pines and Beetles

Lodgepole Pine

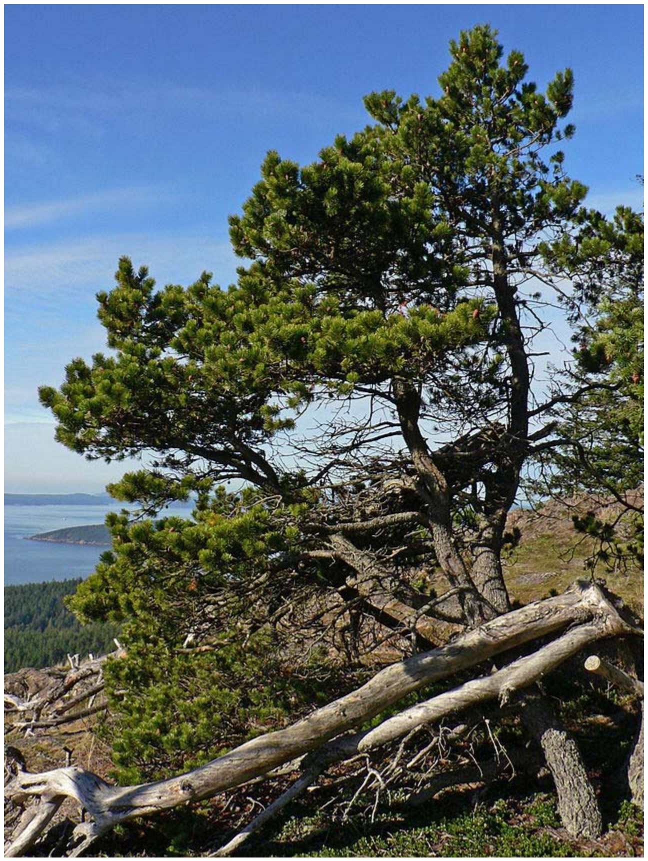


Photo by Walter Siegmund

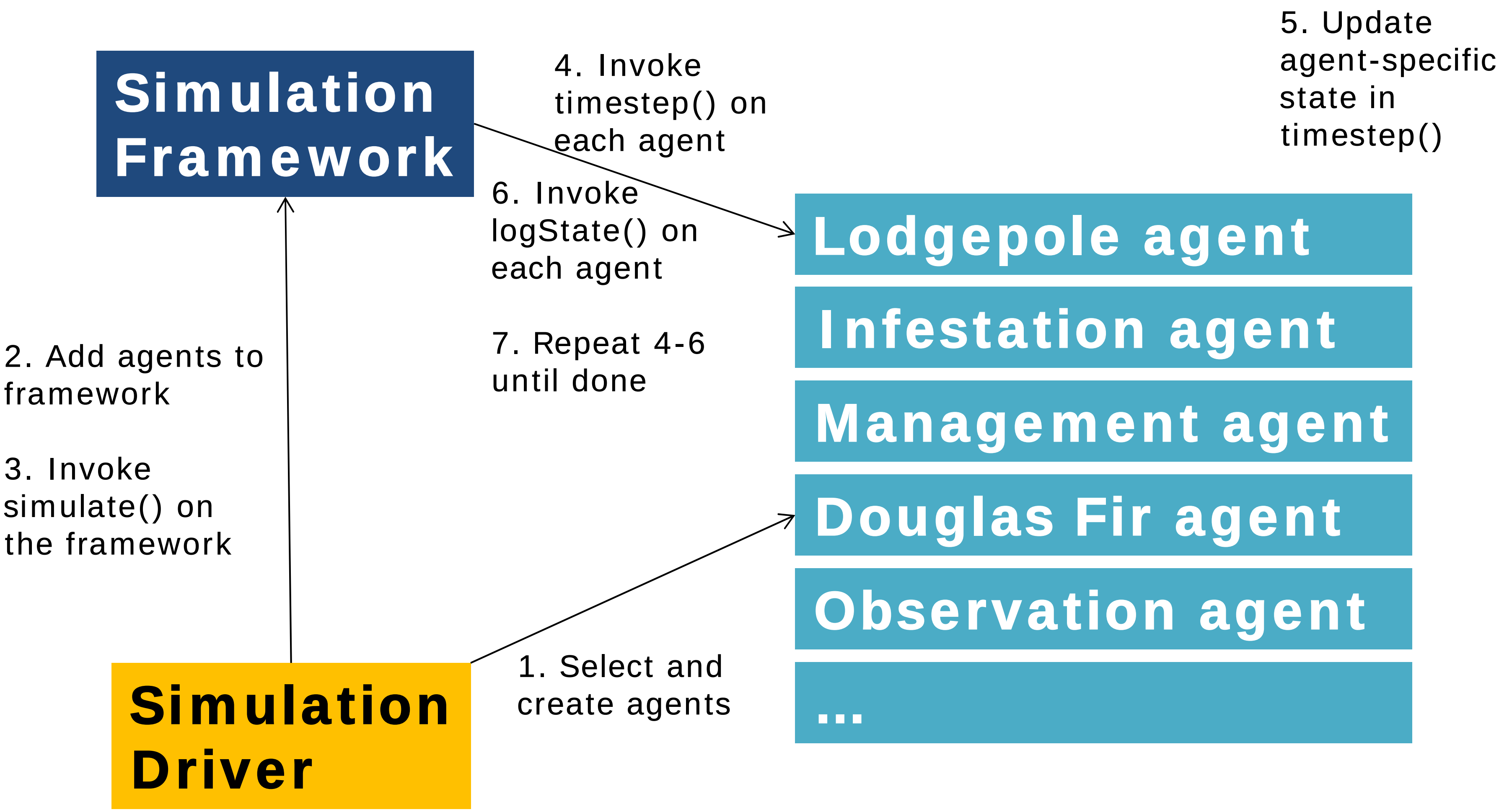
Mountain Pine Beetle

Galleries carved in inner bark

Widespread tree death

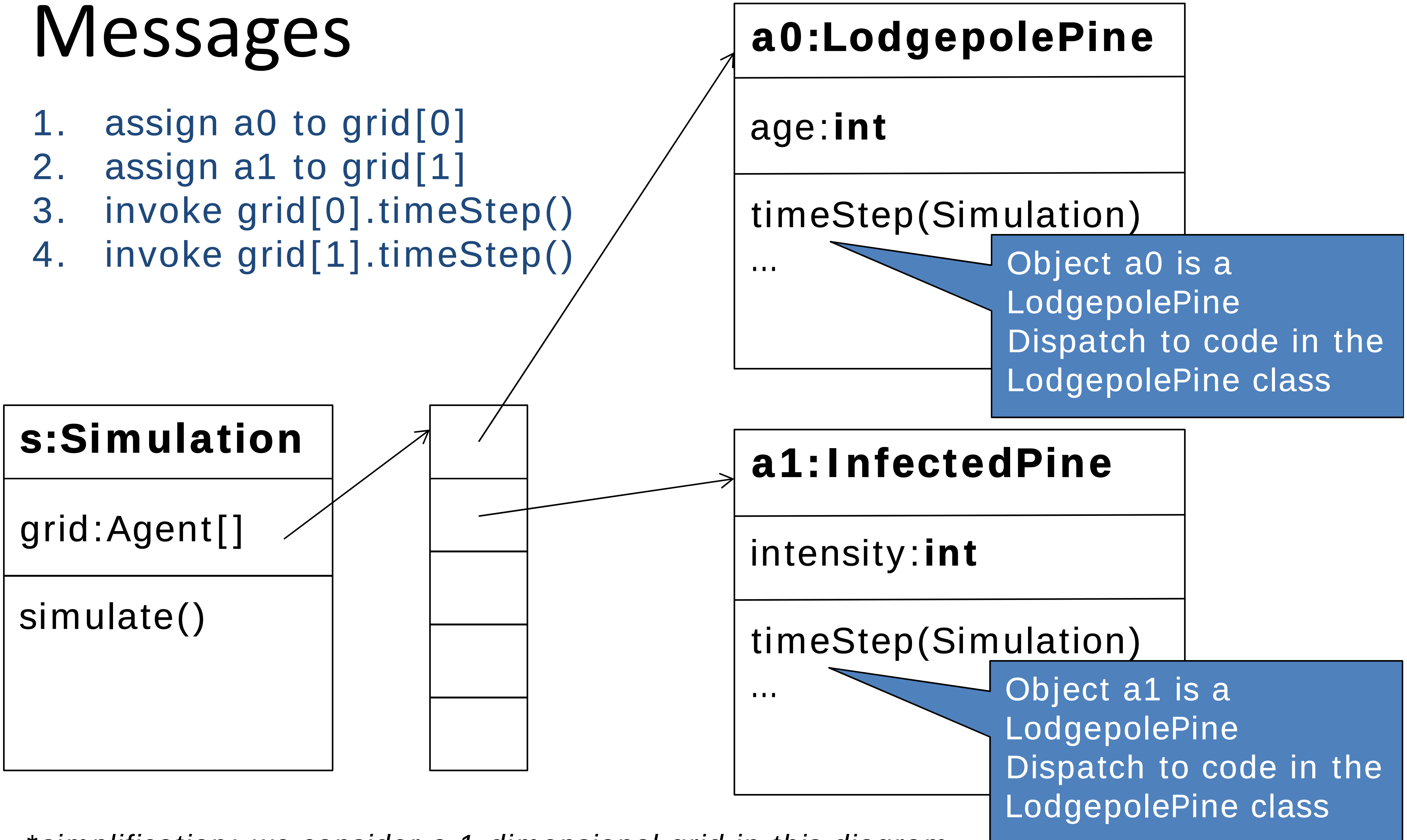
Further reading: Liliana Péreza and Suzana Dragićević. Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations. International Congress on Environmental Modelling and Software Modelling for Environment's Sake, 2010.

Simulation Framework Behavior Model



Today: How Objects Respond to Messages

1. assign a0 to grid[0]
2. assign a1 to grid[1]
3. invoke grid[0].timeStep()
4. invoke grid[1].timeStep()



**simplification: we consider a 1-dimensional grid in this diagram*

Learning Goals

- Explain the need to design for change and design for division of labor
- Understand subtype polymorphism and dynamic dispatch
  - Distinguish between static and run-time type
  - Explain *static* and *instanceof* and their limitations
- Use encapsulation to achieve information hiding
- Define method contracts beyond type signatures
- Explain the concept of design patterns, their ingredients and applications
- Identify applicability of and apply the strategy design pattern
- Write and automate unit tests

Design Goals, Principles, and Patterns

- Design Goals
  - Design for Change
  - Design for Division of Labor
- Design Principles
  - Explicit Interfaces (clear boundaries)
  - Information Hiding (hide likely changes)
- Design Patterns
  - Strategy Design Pattern
  - Composite Design Pattern
- Supporting Language Features
  - Subtype Polymorphism
  - Encapuslation

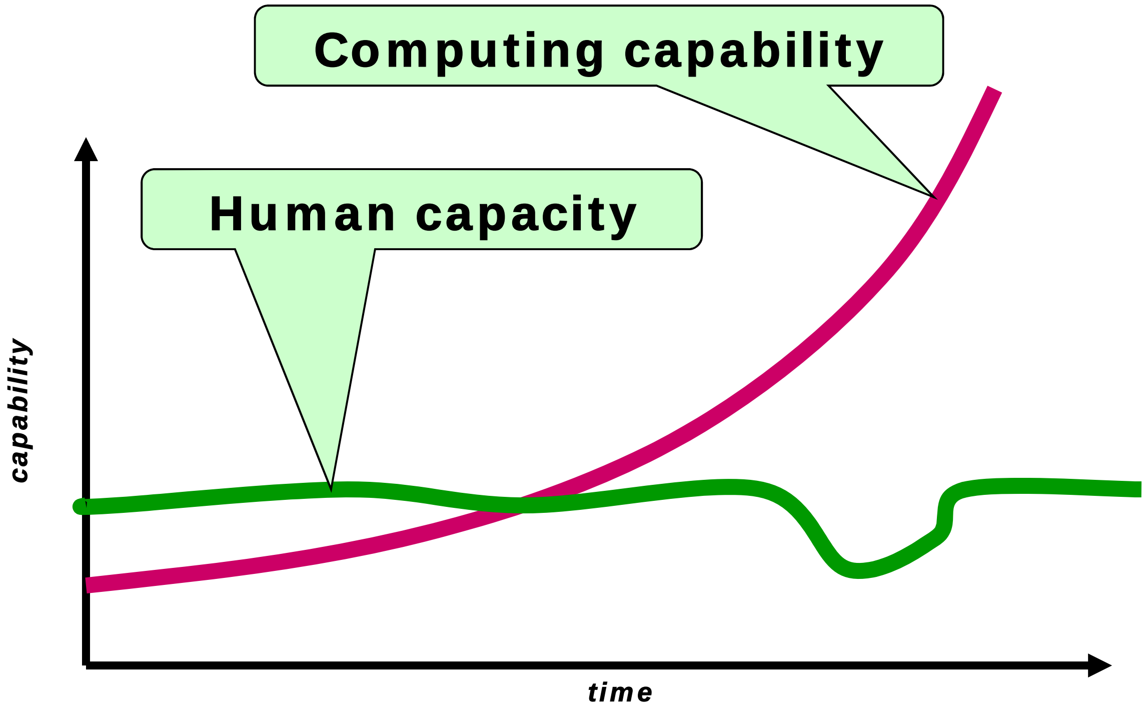
Software Change

- …accept the fact of change as a way of life, rather than an untoward and annoying exception.
  —Brooks, 1974

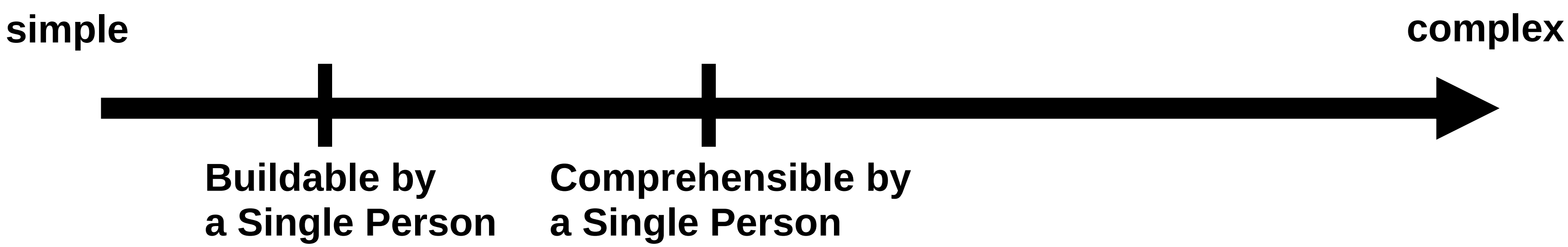
- Software that does not change becomes useless over time.
  —Belady and Lehman

- For successful software projects, most of the cost is spent evolving the system, not in initial development
  - Therefore, reducing the cost of change is one of the most important principles of software design

The limits of exponentials



Building Complex Systems



- Division of Labor
- Division of Knowledge and Design Effort
- Reuse of Existing Implementations

Design Goals for Today

- **Design for Change** (flexibility, extensibility, modifiability)

also

- Design for Division of Labor
- Design for Understandability

SUBTYPE POLYMORPHISM / DYNAMIC DISPATCH (OBJECT-ORIENTED LANGUAGE FEATURE ENABLING FLEXIBILITY)

12

Objects

- A package of state (data) and behavior (actions)
- Can interact with objects by sending messages
  - perform an action (e.g., move)
  - request some information (e.g., getSize)

```
Point p = …
int x = p.getX();
```

```
IntSet a = …; IntSet b = …
boolean s = a.isSubsetOf(b);
```

- Possible messages described through an interface

```
interface Point {
   int getX();
   int getY();
   void moveUp(int y);
   Point copy();
}
```

```
interface IntSet {
   boolean contains(int element);
   boolean isSubsetOf(
                 IntSet otherSet);
}
```

Subtype Polymorphism

- There may be multiple implementations of an interface
- Multiple implementations coexist in the same program
- May not even be distinguishable

- Every object has its own data and behavior

Creating Objects

```
interface Point {
   int getX();
   int getY();
}
Point p = new Point() {
   int getX() { return 3; }
   int getY() { return -10; }
}
```

Classes as Object Templates

```
interface Point {
    int getX();
    int getY();
}
class CartesianPoint implements Point {
    int x,y;
    CartesianPoint(int x, int y) {this.x=x; this.y=y;}
    int getX() { return this.x; }
    int getY() { return y; }
}
Point p = new CartesianPoint(3, -10);
```

More Classes

```
interface Point {
    int getX();
    int getY();
}
class SkewedPoint implements Point {
    int x,y;
    SkewedPoint(int x, int y) {this.x=x + 10; this.y=y * 2;}
    int getX() { return this.x  -  10; }
    int getY() { return this.y / 2; }
}
Point p = new SkewedPoint(3, -10);
```

Polar Points

```
interface Point {
    int getX();
    int getY();
}
class PolarPoint implements Point {
    double len, angle;
    PolarPoint(double len, double angle)
        {this.len=len; this.angle=angle;}
    int getX() { return this.len * cos(this.angle);}
    int getY() { return this.len * sin(this.angle); }
    double getAngle() {…}
}
Point p = new PolarPoint(5, 0.245);
```

Polar Points

```
interface Point {
    int getX();
    int getY();
}
```

```
interface PolarPoint {
    double getAngle() ;
    double getLength();
}
```

```
class PolarPointImpl implements Point, PolarPoint {
    double len, angle;
    PolarPoint(double len, double angle)
        {this.len=len; this.angle=angle;}
    int getX() { return (int) this.len * cos(this.angle);}
    int getY() { return (int) this.len * sin(this.angle); }
    double getAngle() {…}
    double getLength() {… }
}
PolarPoint p = new PolarPointImpl(5, 0.245);
Point q = new PolarPointImpl(5, 0.245);
```

Middle Points

```
interface Point {
    int getX();
    int getY();
}
class MiddlePoint implements Point {
    Point a, b;
    MiddlePoint(Point a, Point b) {this.a = a; this.b = b; }
    int getX() { return (this.a.getX() + this.b.getX()) / 2;}
    int getY() { return (this.a.getY() + this.b.getY()) / 2; }
}
Point p = new MiddlePoint(new PolarPoint(5, 0.245),
                          new CartesianPoint(3, 3));
```

Example: Points and Rectangles

Subtype Polymorphism

```
interface Point {
    int getX();
    int getY();
}
… = new Rectangle() {
    Point origin = …;
    int width = …;
    int height = …;
    Point getOrigin() { return this.origin; }
    int getWidth() { return this.width; }
    void draw() {
        this.drawLine(this.origin.getX(), this.origin.getY(),      // first line
                this.origin.getX()+this.width, this.origin.getY());
        … // more lines here
    }
};
```

Points and Rectangles: Interface

```
interface Point {
    int getX();
    int getY();
}
interface Rectangle {
    Point getOrigin();
    int getWidth();
    int getHeight();
    void draw();
}
```

What are possible implementations of the Rectangle interface?

Java interfaces and classes

- Organize program functionality around kinds of abstract "objects"
  - For each object kind, offer a specific set of operations on the objects
  - Objects are otherwise opaque: Details of representation are hidden
  - "Messages to the receiving object"
- Distinguish interface from class
  - Interface: expectations
  - Class: delivery on expectations (the implementation)
  - Anonymous class: special Java construct to create objects without explicit classes: Point x = **new** Point() { /* implementation */ };
- Explicitly represent the taxonomy of object types
  - This is the type hierarchy (!= inheritance, more on that later): A CartesianPoint is a Point

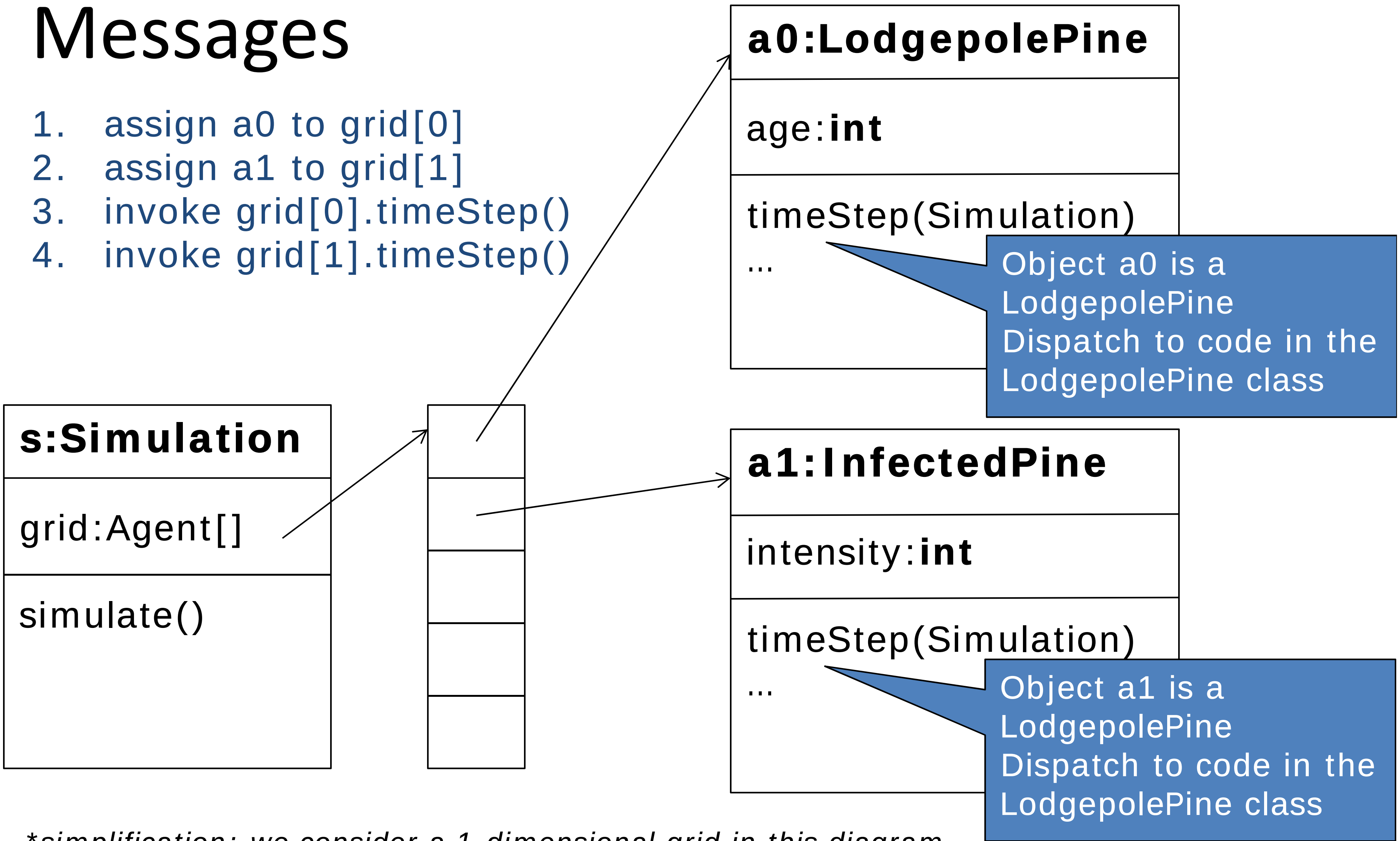
Discussion: Subtype Polymorphism

- A user of an object does not need to know the object's implementation, only its interface
- All objects implementing the interface can be used interchangeably
- Allows flexible **change** (modifications, extensions, reuse) later without changing the client implementation, even in unanticipated contexts

Design for Change!

Today: How Objects Respond to Messages

1. assign a0 to grid[0]
2. assign a1 to grid[1]
3. invoke grid[0].timeStep()
4. invoke grid[1].timeStep()



**simplification: we consider a 1-dimensional grid in this diagram*

Check your Understanding

```
interface Animal {
    void makeSound();
}
class Dog implements Animal {
    public void makeSound() { System.out.println("bark!"); }
}
class Cow implements Animal {
    public void makeSound() { mew(); }
    public void mew() {System.out.println("Mew!"); }
}
0 Animal x = new Animal() {
        public void makeSound() { System.out.println("chirp!"); }};
  x.makeSound();
1 Animal a = new Animal();
2 a.makeSound();
3 Dog d = new Dog();
4 d.makeSound();
5 Animal b = new Cow();
6 b.makeSound();
7 b.mew();
```

- What happens?

Historical Note: Simulation and the Origins of Objects

Dahl and Nygaard at the time of Simula's development

- Simula 67 was the first object-oriented programming language
- Developed by Kristin Nygaard and Ole-Johan Dahl at the Norwegian Computing Center
- Developed to support discrete-event simulations
  - Much like our tree beetle simulation
  - Application: operations research, e.g. for traffic analysis
  - Extensibility was a key quality attribute for them
  - Code reuse was another—which we will examine later

See textbook 26.7

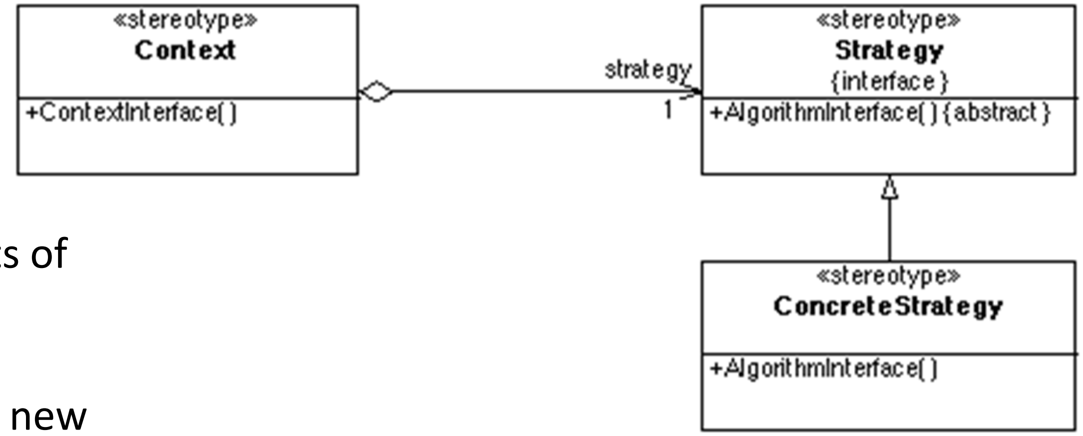
STRATEGY DESIGN PATTERN (EXPLOITING POLYMORPHISM FOR FLEXIBILITY)

Tradeoffs?

```
void sort(int[] list, boolean inOrder) {
   …
  boolean mustswap;
  if (inOrder) {
    mustswap = list[i] < list[j];
  } else {
    mustswap = list[i] > list[j];
  }
  …
}
```

```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int i, int j) { return i<j; }}

class DownComparator implements Comparator {
  boolean compare(int i, int j) { return i>j; }}
```

Behavioral Pattern: Strategy



- Applicability
  - Many classes differ in only their behavior
  - Client needs different variants of an algorithm
- Consequences
  - Code is more extensible with new strategies
    - compare to conditionals
  - Separates algorithm from context
    - each can vary independently
    - design for change and reuse; reduce coupling
  - Adds objects and dynamism
    - code harder to understand
  - Common strategy interface
    - may not be needed for all Strategy implementations – may be extra overhead

- Design for change
  - Find what varies and encapsulate it
  - Allows changing/adding alternative variations later
  - Class *Context* closed for modification, but open for extension
- Equivalent in functional progr. languages: Higher-order functions
  - But a Strategy interface may include more than one function

Design Patterns

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”
  – Christopher Alexander

- Every Strategy interface has its own domain-specific interface
  - But they share a common problem and solution

Examples

- Change the sorting criteria in a list
- Change the aggregation method for computations over a list (e.g., fold)
- Compute the tax on a sale
- Compute a discount on a sale
- Change the layout of a form

Benefits of Patterns

- Shared language of design
  - Increases communication bandwidth
  - Decreases misunderstandings

- Learn from experience
  - Becoming a good designer is hard
    - Understanding good designs is a first step
  - Tested solutions to common problems
    - Where is the solution applicable?
    - What are the tradeoffs?

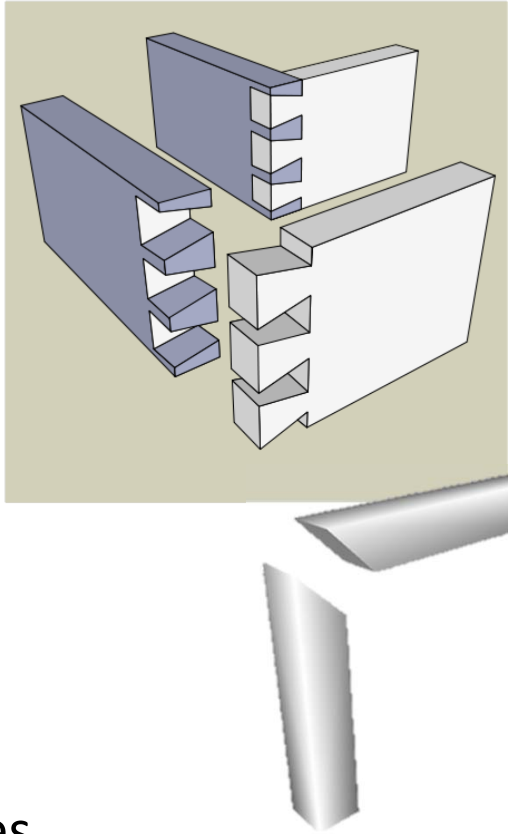
Illustration [Shalloway and Trott]

- Carpenter 1: How do you think we should build these drawers?

- Carpenter 2: Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating…

- SE example: “I wrote this if statement to handle … followed by a while loop … with a break statement so that…”

A Better Way

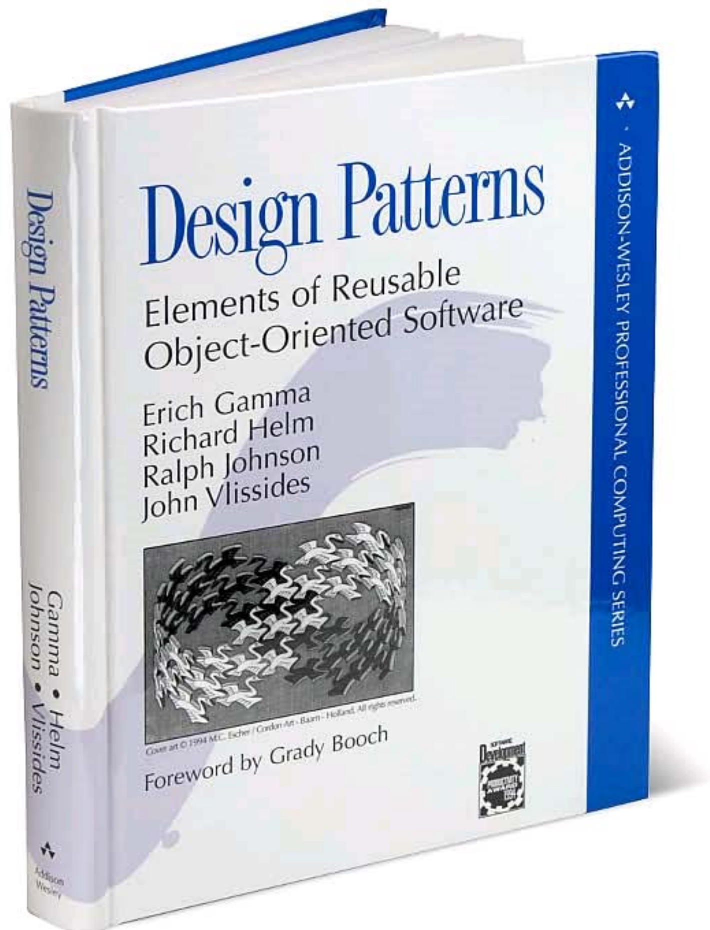


- Carpenter 1: Should we use a dovetail joint or a miter joint?
- Subtext:
  - miter joint: cheap, invisible, breaks easily
  - dovetail joint: expensive, beautiful, durable
- Shared terminology and knowledge of consequences raises level of abstraction
  - CS: Should we use a Strategy?
  - Subtext
    - Is there a varying part in a stable context?
    - Might there be advantages in limiting the number of possible implementations?

Elements of a Pattern

- Name
  - Important because it becomes part of a design vocabulary
  - Raises level of communication
- Problem
  - When the pattern is applicable
- Solution
  - Design elements and their relationships
  - Abstract: must be specialized
- Consequences
  - Tradeoffs of applying the pattern
    - Each pattern has costs as well as benefits
    - Issues include flexibility, extensibility, etc.
    - There may be variations in the pattern with different consequences

History: Design Patterns Book

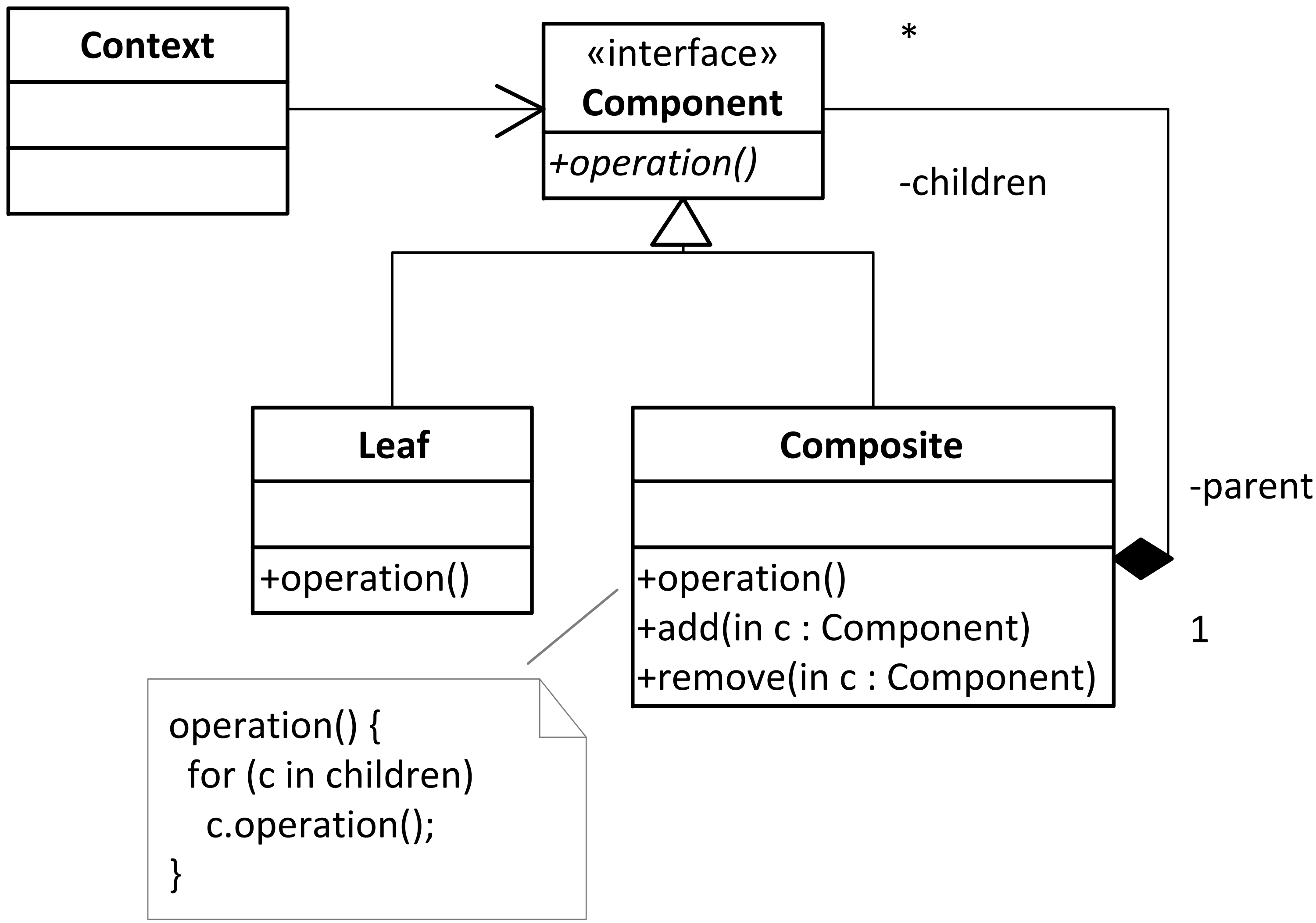


- Brought Design Patterns into the mainstream
- Authors known as the Gang of Four (GoF)
- Focuses on *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*
- Great as a reference text
- Uses C++, Smalltalk

Design Exercise (on paper)

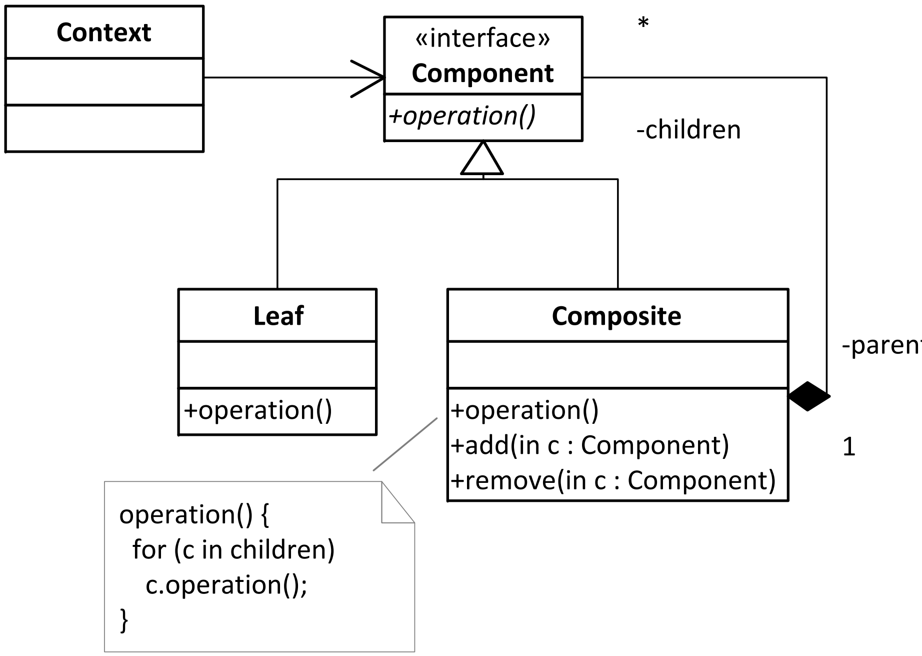
- You are designing software for a shipping company.
- There are several different kinds of items that can be shipped: letters, books, packages, fragile items, etc.
- Two important considerations are the **weight** of an item and its **insurance cost**.
  - Fragile items cost more to insure.
  - All letters are assumed to weigh an ounce
  - We must keep track of the weight of other packages.
- The company sells **boxes** and customers can put several items into them.
  - The software needs to track the contents of a box (e.g. to add up its weight, or compute the total insurance value).
  - However, most of the software should treat a box holding several items just like a single item.
- Think about how to represent packages; what are possible interfaces, classes, and methods? (letter, book, box only)

The Composite Design Pattern



The Composite Design Pattern

- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components



We have seen this before

```
interface Point {
        int getX();
        int getY();
}
class MiddlePoint implements Point {
        Point a, b;
        MiddlePoint(Point a, Point b) {this.a = a; this.b = b; }
        int getX() { return (this.a.getX() + this.b.getX()) / 2;}
        int getY() { return (this.a.getY() + this.b.getY()) / 2; }
}
```

ENCAPSULATION (LANGUAGE FEATURE TO CONTROL VISIBILITY)

Controlling Access – Best practices

- Define an interface
- Client may only use the messages in the interface
- Fields not accessible from client code
- Methods only accessible if exposed in interface

```
interface Point {
    int getX();
    int getY();
}
class CartesianPoint implements Point {
    int x,y;
    Point(int x, int y) {this.x=x; this.y=y;}
    int getX() { return this.x; }
    int getY() { return this.y; }
    String getText() { return this.x + " x " + this.y; }
}
Point p = new CartesianPoint(3, -10);
p.getX();
p.getText(); // not accessible
p.x; // not accessible
```

Interface Type

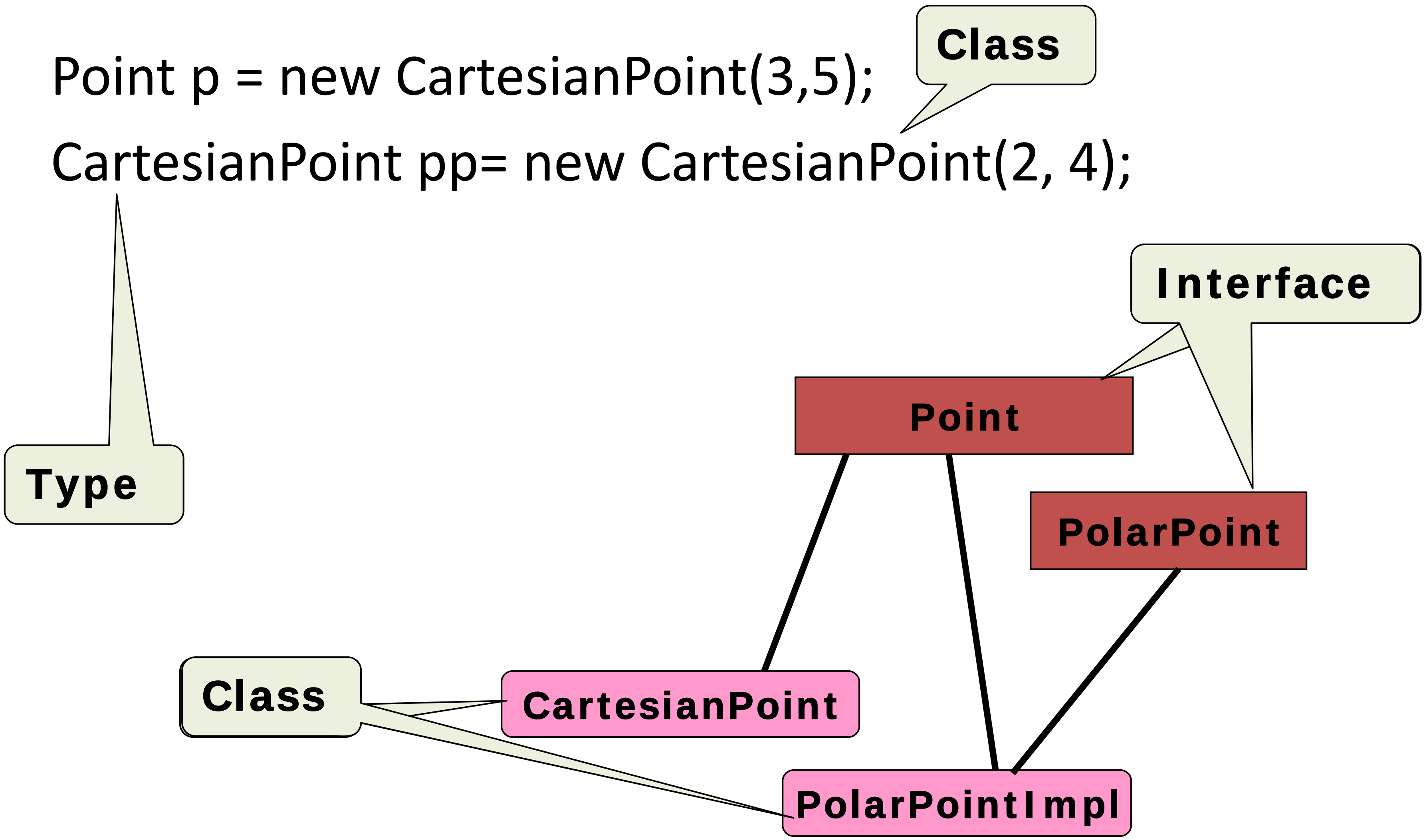
Java: Classes as Types

- Classes usable as type
  - (Public) methods in classes usable like methods in interfaces
  - (Public) fields directly accessible from other classes
  - Language constructs (public, private, protected) to control access
- Prefer programming to interfaces (variables should have **interface type**, not class type)
  - Esp. whenever there are multiple implementations of a concept
  - Supports changing to different implementations later
  - Prevents dependence on implementation details

```
int add(CartesianPoint p) { …   // preferably no
int add(Point p) { …            // yes!
```

Interfaces vs Classes as Types

```
Point p = new CartesianPoint(3,5);
CartesianPoint pp= new CartesianPoint(2, 4);
```



Interfaces and Classes (Review)

```
class PolarPoint implements Point {
    double len, angle;
    PolarPoint(double len, double angle)
            {this.len=len; this.angle=angle;}
    int getX() { return this.len * cos(this.angle);}
    int getY() { return this.len * sin(this.angle); }
    double getAngle() { return angle; }
}
```

```
Point p = new PolarPoint(5, .245);
p.getX();
p.getAngle(); // not accessible
p.len // not accessible
```

```
PolarPoint pp = ...
pp.getX();
pp.getAngle();
pp.len
```

Java: Visibility Modifiers

```
class Point {
	private int x, y;
	public int getX() { return this.x; } // a method; getY() is similar
	public Point(int px, int py) { this.x = px; this.y = py; }// constructor creating the object
}
class Rectangle {
	private Point origin;
	private int width, height;
	public Point getOrigin() { return origin; }
	public int getWidth() { return width; }
	public void draw() {
		drawLine(this.origin.getX(), this.origin.getY(),	// first line
				this.origin.getX()+this.width, origin.getY());
		… // more lines here
	}
	public Rectangle(Point o, int w, int h) {
		this.origin = o; this.width = w; this.height = h;
	}
}
```

Hiding interior state

```
class Point {
    private int
    public int
    public Poi
}
class Rectang
    private Poi
    private int
    public Point getOrigin() { return origin; }
    public int getWidth() { return width; }
    public voi
        o
        ..
    }
    public Rec
        o
    }
}
```

Some Client Code

```
Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5
```

Client Code that will *not* work in this version

```
Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.origin.x + r.width; //  trying to "look inside"
```

Hiding interior state

```
class Point {
    private int
    public int
    public Poi
}
class Rectang
    private Po
    private int
    public Point getOrigin() { return origin; }
    public int getWidth() { return width; }
    public void draw() {
        drawLine(origin.getX(), origin.getY(),          // first line
                 origin.getX()+width, origin.getY());
        … // more lines here
    }
    public Rectangle(Point o, int w, int h) {
        origin = o; width = w; height = h;
    }
}
```

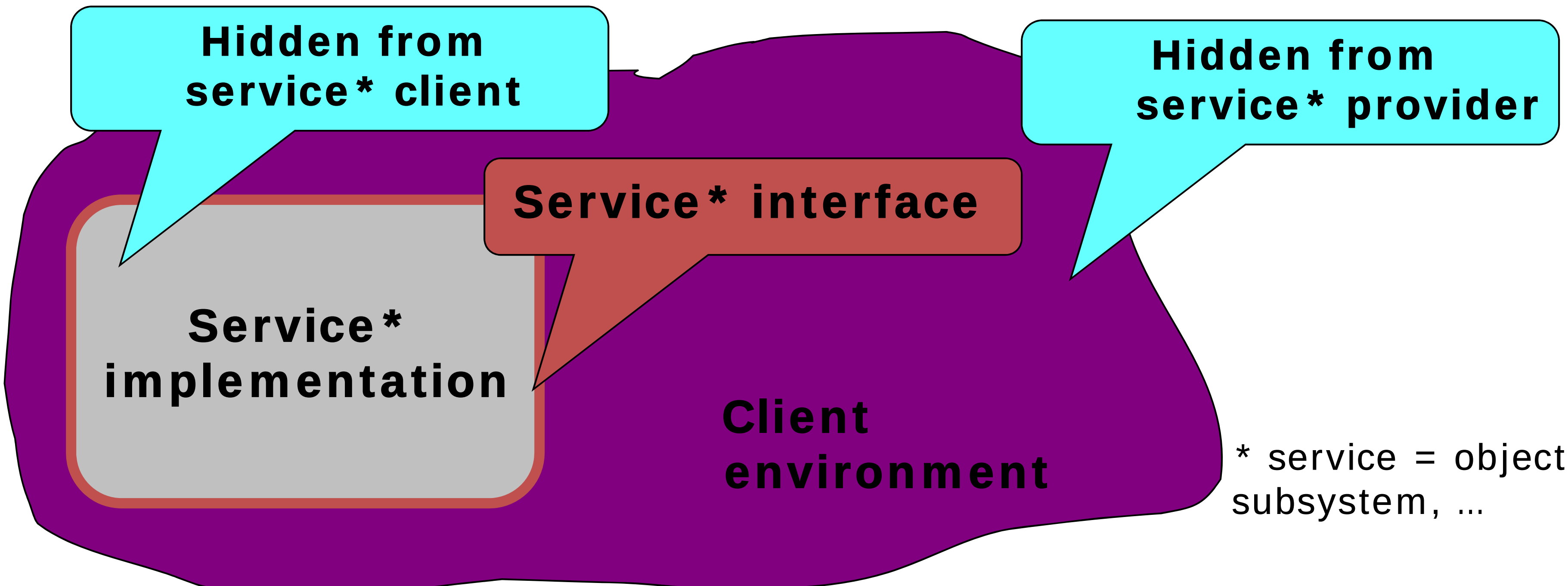
Discussion:

- ***What are the benefits of* private *fields?***
- ***Methods can also be private – why is this useful?***

DESIGN PRINCIPLE: INFORMATION HIDING

Fundamental Design Principle for Change: Information Hiding

- Expose as little implementation detail as necessary
- Allows to change hidden details later



* service = object, subsystem, …

Information Hiding

- Interfaces (contracts) remain stable
- Hidden implementation can be changed easily
- => Identify what is likely to change, and hide it
- => Requires anticipation of change (judgment)

- Points example: Minimal stable interface, allows alternative implementations and flexible composition

- (Not all change can be anticipated, causing maintenance work or reducing flexibility)

Micro-Scale Information Hiding

- How could we better hide information here?

```
class Utilities {
    private int total;
    public int sum(int[] data) {
        total = 0;
        for (int i = 0; i < data.length; ++i) {
            total += data[i];
        }
        return total
    }
    // other methods…
}
```

- Should be a local variable of the sum method
- This would hide part of the implementation of sum from the rest of the class!

A Great Piazza Question

- **Should I add a getter/setter for every private field in a class?**
  - What do you think?

54

A Great Piazza Question

- **Should I add a getter/setter for every private field in a class?**
  - What do you think?
- Information hiding suggests including:
  - A getter only when clients need the information
  - A setter only when clients need to mutate the data
    - Avoid where possible!
  - Methods with signatures at the right level of abstraction

55

An Infamous Design Problem

```
// Represents a Java class
class Class {
        // Entities that have digitally signed this class
        // so use the only class if you trust a signer
        private Object[] signers;

        // what getters/setters should we provide?



}
```


56

An Actual* (Insecure!) Design

```
// Represents a Java class
class Class {
        // Entities that have digitally signed this class
        // so use the only class if you trust a signer
        private Object[] signers;

        // Get the signers of this class
        // VULNERABILITY: clients can change
        // the signers of a class
        public Object[] getSigners() { return signers; }
}
```

*simplified slightly for presentation, but a real Java bug (now fixed)

A Better*† Design – Abstract and Immutable

```
// Represents a Java class
class Class {
    // Entities that have digitally signed this class
    private Object[] signers;

    // Get the signers of this class
    public List getSigners() {
        List signerList = Arrays.asList(signers);
        return Collections.unmodifiableList(signerList);
    }
}
```

*sadly not used in Java; they had to keep the poorly designed signature for compatibility, but the code makes a copy of the array so it is secure
†even better (performance-wise) would be to store the signers in an unmodifiable list, putting the wrapper calls in the Class constructor

Information Hiding promotes Reuse

- Think in terms of abstractions not implementations
  - e.g., Point vs CartesianPoint
- Abstractions can often be reused
- Different implementations of the same interface possible,
  - e.g., reuse Rectangle but provide different Point implementation
- Decoupling implementations
- Hiding internals of implementations

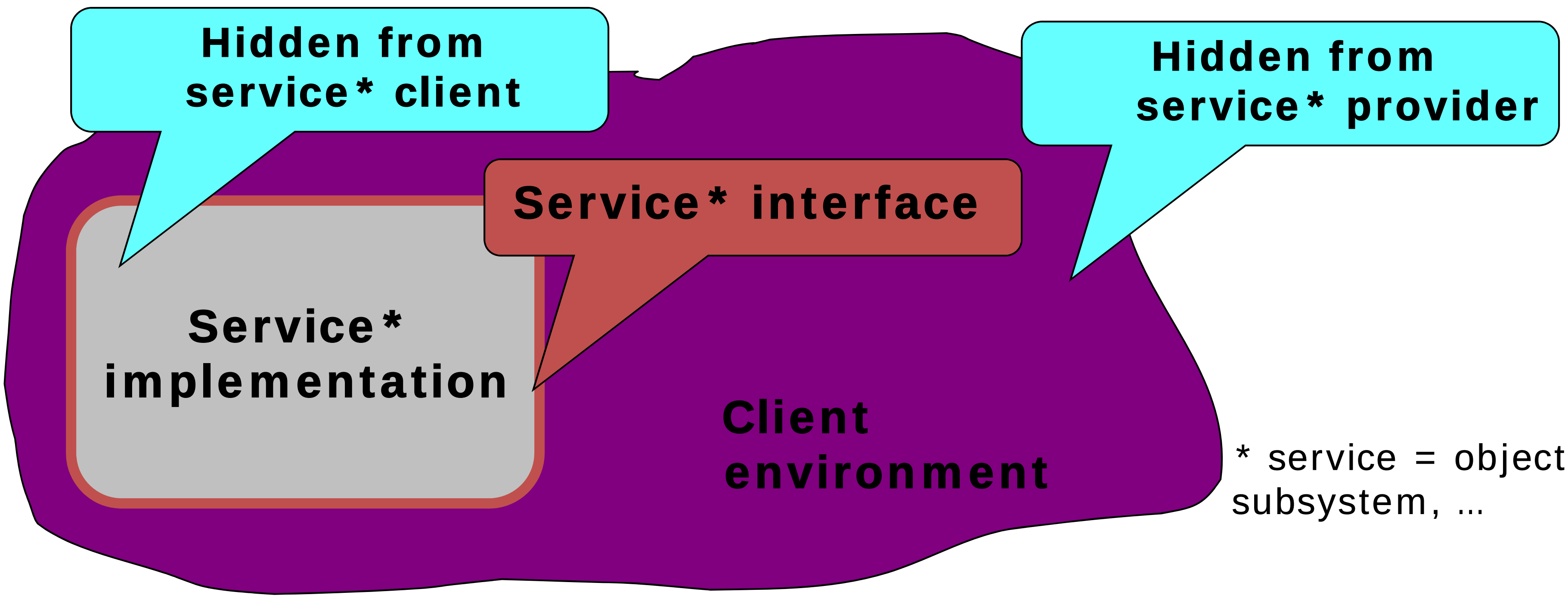
More on reuse next week

INFORMATION HIDING CASE STUDY

Agents, KWIC

CONTRACTS
(BEYOND TYPE SIGNATURES)

Contracts and Clients



* service = object, subsystem, …

Contracts

- Agreement between provider and users of an object
- Includes
  - Interface specification
  - Functionality and correctness expectations
  - Performance expectations

Who's to blame?

```
Algorithms.shortestDistance(g,
   "Tom", "Anne");

> ArrayOutOfBoundsException
```

Who’s to blame?

```
Algorithms.shortestDistance(g,
   “Tom”, “Anne”);

> -1
```

Who's to blame?

```
Algorithms.shortestDistance(g,
   "Tom", "Anne");

> 0
```

Who’s to blame?

```
class Algorithms {
    /**
     * This method finds the
     * shortest distance between to
     * verticies. It returns -1 if
     * the two nodes are not
     * connected. */
    int shortestDistance(…) {…}
}
```

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let *k* be the number of bytes actually read; these bytes will be stored in elements b[off] throughb[off+*k*-1], leaving elements b[off+*k*] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- Throws:
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data f
  attempt is made to read as ma
  The number of bytes actually r
  until input data is available, en
- If len is zero, then no bytes ar
  attempt to read at least one by
  end of file, the value -1 is retu
  into b.
- The first byte read is stored in
  on. The number of bytes read
  bytes actually read; these byte
  1], leaving elements b[off+*k*]
- In every case, elements b[0] through b[off] and
  elements b[off+len] through b[b.length-1] are unaffected.

- Specification of return
- Timing behavior (blocks)
- Case-by-case spec
  - len=0 → return 0
  - len>0 && eof → return -1
  - len>0 && !eof →return >0
- Exactly where the data is stored
- What parts of the array are not affected

- Throws:
  - IOException - If the first byte c
    or if the input stream has been
  - NullPointerException - If b is n
  - IndexOutOfBoundsException -
    than b.length - off

- Multiple error cases, each with a precondition
- Includes “runtime exceptions” not in throws clause

Specifications

- Contains
  - Functional behavior
  - Erroneous behavior
  - Quality attributes (performance, scalability, security, …)
- Desirable attributes
  - Complete
    - Does not leave out any desired behavior
  - Minimal
    - Does not require anything that the user does not care about
  - Unambiguous
    - Fully specifies what the system should do in every case the user cares about
  - Consistent
    - Does not have internal contradictions
  - Testable
    - Feasible to objectively evaluate
  - Correct
    - Represents what the end-user(s) need

Function Specifications

- A function’s contract is a statement of the responsibilities of that function, and the responsibilities of the code that calls it.
  - Analogy: legal contracts - If you pay me $30,000, I will build a new room on your house
  - Helps to pinpoint responsibility
- Contract structure
  - Precondition: the condition the function relies on for correct operation
  - Postcondition: the condition the function establishes after correctly running
- (Functional) correctness with respect to the specification
  - If the client of a function fulfills the function’s precondition, the function will execute to completion and when it terminates, the postcondition will be fulfilled
- What does the implementation have to fulfill if the client violates the precondition?

What’s wrong with these spec(s)?

```
/** Keeps a running total of a set of values added */
class Accumulator {
        private int total;

        /** Adds a value to the running total */
        public void add(int nextVal) {
                total += nextVal;
        }

        /** Returns the total value added up */
        public int getTotal() {
                int totalValue = total;
                total = 0; // reset for the next
                return totalValue;
        }
}
```

- This method has an undocumented side effect!
- Surprising due to misleading name
- We could document it, but better to provide a separate reset() function
- **Command-Query Separation design rule:** each method should be a *query* or a *command*, never both. [Meyer]

Formal Specifications

```
/*@ requires len >= 0 && array != null && array.length == len;
  @
  @ ensures \result ==
  @            (\sum int j;  0 <= j && j < len;  array[j]);
  @*/
int total(int array[], int len);
```

Advantage of formal specifications:

* runtime checks (almost) for free
* basis for formal verification
* assisting automatic analysis tools

JML (Java Modelling Language) as specifications language in Java (inside comments)

Runtime Checking of Specifications with Assertions

```
/*@ requires len >= 0 && array.length == len
  @ ensures \result ==
  @              (\sum int j;  0 <= j && j < len;  array[j])
  @*/
float sum(int array[], int len) {
    assert len >= 0;
    assert array.length == len;
    float sum = 0.0;
    int i = 0;
    while (i < len) {
        sum = sum + array[i]; i = i + 1;
    }
    assert sum ...;
    return sum;
}
```

java -ea Main

Runtime Checking with Exceptions

```
/*@ requires len >= 0 && array.length == len
  @ ensures \result ==
  @           (\sum int j;  0 <= j && j < len;  array[j])
  @*/
float sum(int array[], int len) {
    if (len < 0 || array.length != len)
        throw IllegalArgumentException(…);
    float sum = 0.0;
    int i = 0;
    while (i < len) {
        sum = sum + array[i]; i = i + 1;
    }
    return sum;
    assert …;
}
```

Check arguments even when assertions are disabled.
Good for robust libraries!

Write a Specification

- Write
  - a type signature,
  - a textual specification, and
  - a formal specification

  for a function **slice(list, from, until)** that returns all values of a list between positions <from> and <until> as a new list

Contacts and Interfaces

- All objects implementing an interface must adhere to the interface's contracts
  - Objects may provide different implementations for the same specification
  - Subtype polymorphism: Client only cares about interface, not about the implementation

p.getX() **s.read()**

=> **Design for Change**

78

Specifications in Practice

- Describe expectations beyond the type signature
- Textual specifications are most common in practice
  - Formal specifications are useful but costly to write

- Advice
  - Write precise specs – even if informal
  - Think in terms of pre- and post-conditions
  - Focus effort on code that is reused or integrated into a bigger system

ASIDE: CONSTRUCTORS AND CLASS INVARIANTS

Data Structure Invariants (cf. 122)

```
struct list {
    elem data;
    struct list* next;
};
struct queue {
    list front;
    list back;
};

bool is_queue(queue Q) {
    if (Q == NULL) return false;
    if (Q->front == NULL || Q->back == NULL) return false;
    return is_segment(Q->front, Q->back);
}
```

Data Structure Invariants (cf. 122)

- Properties of the Data Structure
- Should always hold before and after method execution
- May be invalidated temporarily during method execution

```
void enq(queue Q, elem s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{ … }
```

Class Invariants

- Properties about the fields of an object
- Established by the **constructor**
  - A special method for object initialization
  - Same name as class, no return type
  - If no constructor is written, a no-argument constructor is generated
    - Initializes fields to default values
- Should hold before and after execution of public methods
- May be invalidated temporarily during method execution

Class Invariants and Constructors

```
class Link {
    // default constructor generated
    public int data;  // ok to be public; Link is internal to Queue
    public Link next;
};
public class Queue {
    private Link front;
    private Link back;
    public Queue() {
        front = new Link();  // calls the default constructor for Link
        back = front;
        assert isQueue();    // the invariant should hold now!
    }
    public boolean isQueue() {
        if (front == null || back == null) return false;
        return isSegment(front, back);
    }
    private boolean isSegment(Link first, Link last) { … }
};
```

FUNCTIONAL CORRECTNESS (UNIT TESTING AGAINST INTERFACES)

Context

- **Design for Change** as goal
- **Encapsulation** provides technical means
- **Information Hiding** as design strategy
- **Contracts** describe behavior of hidden details
- **Testing** helps gaining confidence in functional correctness (w.r.t. contracts)

Functional Correctness

- The compiler ensures that the types are correct (type checking)
  - Prevents “Method Not Found” and “Cannot add Boolean to Int” errors at runtime
- Static analysis tools (e.g., FindBugs) recognize certain common problems
  - Warns on possible NullPointerExceptions or forgetting to close files
- How to ensure functional correctness of contracts beyond type correctness and bug patterns?

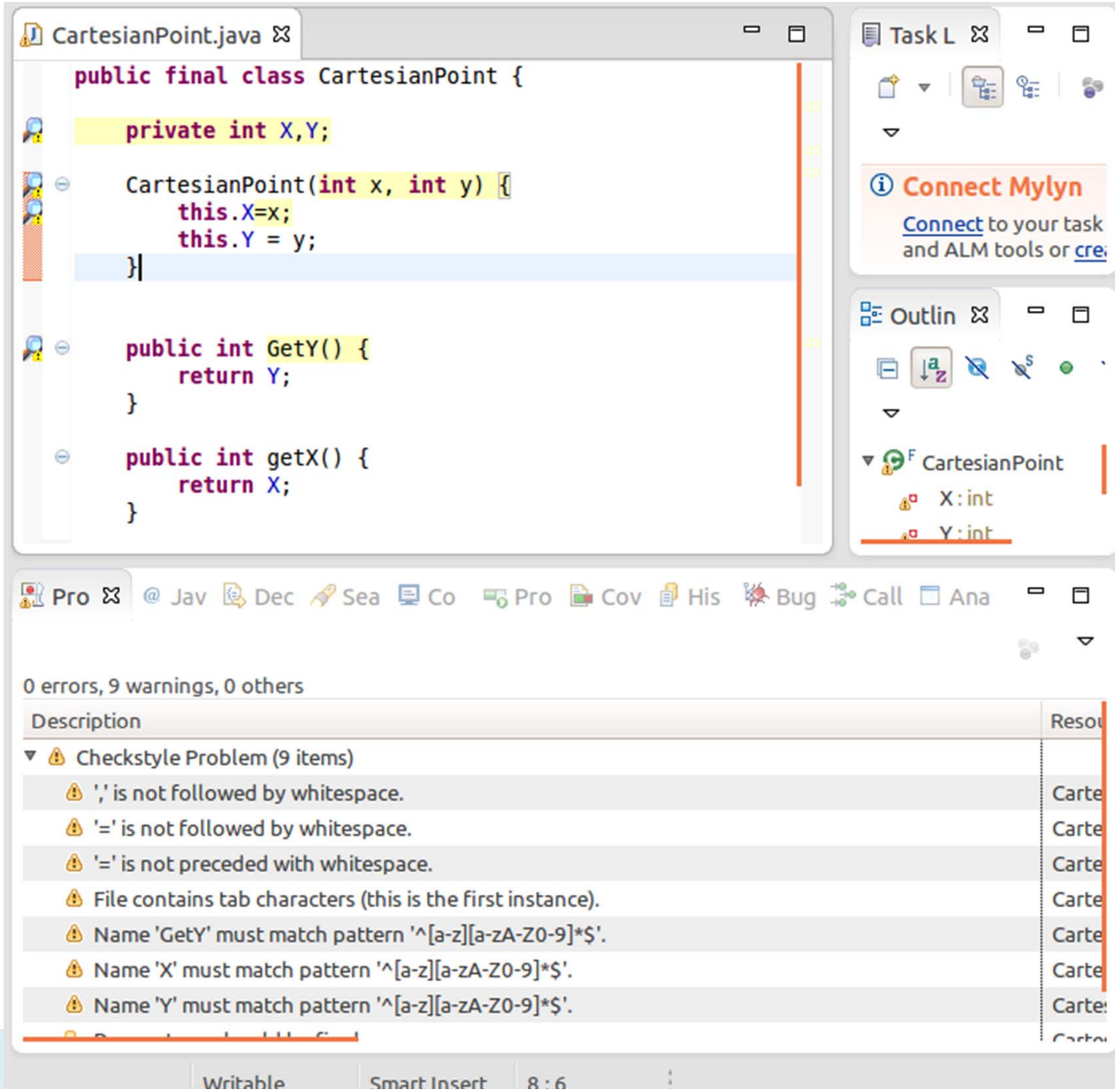
Type Checking Example

```
interface Animal {
    void makeSound();
}
class Dog implements Animal {
    public void makeSound() { System.out.println("bark!"); }
}
class Cow implements Animal {
    public void makeSound() { mew(); }
    public void mew() {System.out.println("Mew!"); }
}
```

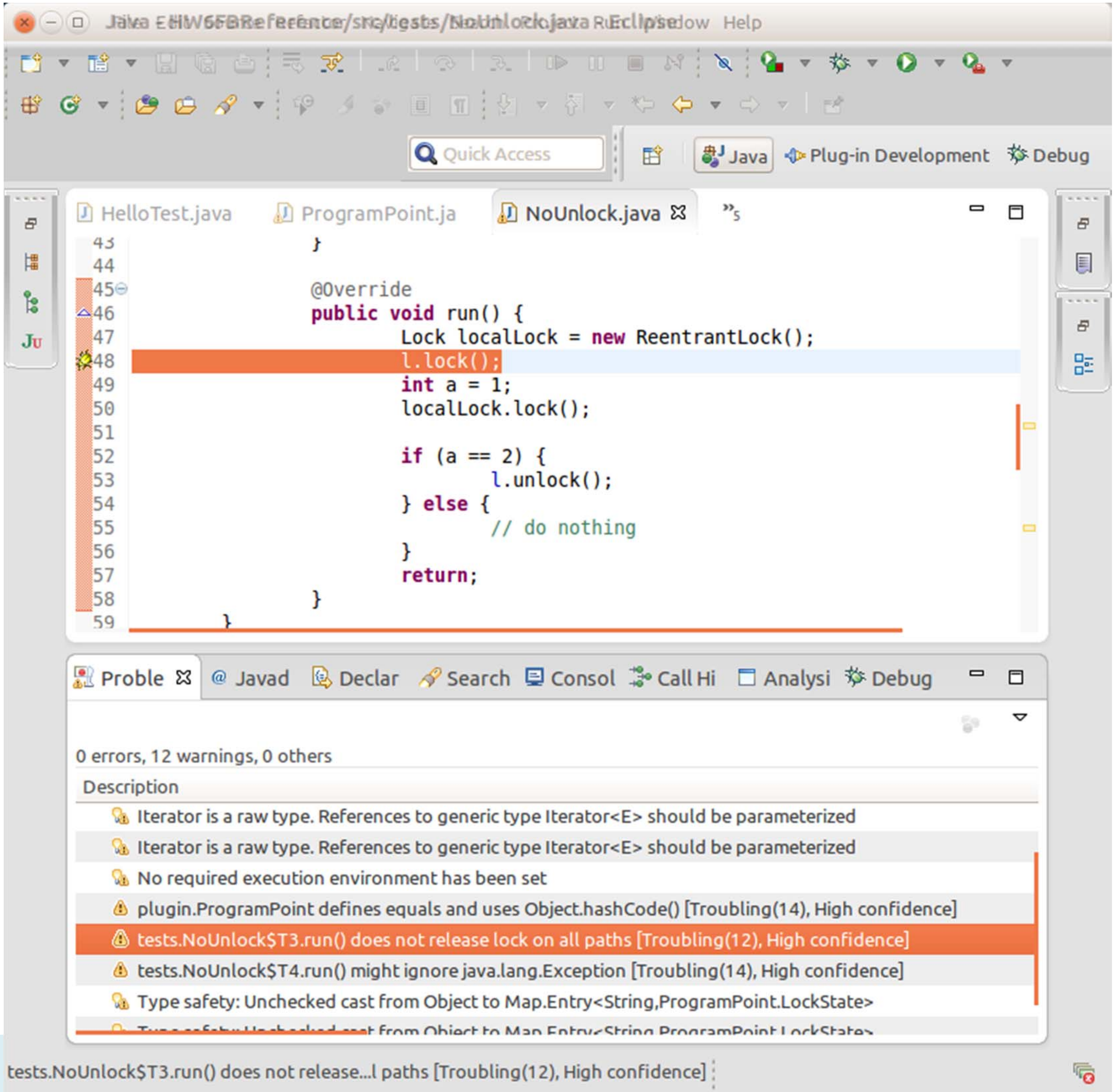
```
Animal a = new Animal();
a.makeSound();
Dog d = new Dog();
d.makeSound();
Animal b = new Cow();
b.mew();
b.jump();
```

- What happens?

CheckStyle


CartesianPoint.java
public final class CartesianPoint {
private int X,Y;
CartesianPoint(int x, int y) {
this.X=x;
this.Y = y;
}
public int GetY() {
return Y;
}
public int getX() {
return X;
}
Task L
Connect Mylyn
Connect to your task
and ALM tools or crea
Outlin
CartesianPoint
X : int
Y : int
Pro
Jav
Dec
Sea
Co
Pro
Cov
His
Bug
Call
Ana
0 errors, 9 warnings, 0 others
Description
Resou
Checkstyle Problem (9 items)
',' is not followed by whitespace.
'=' is not followed by whitespace.
'=' is not preceded with whitespace.
File contains tab characters (this is the first instance).
Name 'GetY' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'X' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'Y' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Writable
Smart Insert
8 : 6

FindBugs



Quick Access
Java
Plug-in Development
Debug
HelloTest.java
ProgramPoint.ja
NoUnlock.java
@Override
public void run() {
Lock localLock = new ReentrantLock();
l.lock();
int a = 1;
localLock.lock();
if (a == 2) {
l.unlock();
} else {
// do nothing
}
return;
}
Proble
Javad
Declar
Search
Consol
Call Hi
Analysi
Debug
0 errors, 12 warnings, 0 others
Description
Iterator is a raw type. References to generic type Iterator<E> should be parameterized
Iterator is a raw type. References to generic type Iterator<E> should be parameterized
No required execution environment has been set
plugin.ProgramPoint defines equals and uses Object.hashCode() [Troubling(14), High confidence]
tests.NoUnlock$T3.run() does not release lock on all paths [Troubling(12), High confidence]
tests.NoUnlock$T4.run() might ignore java.lang.Exception [Troubling(14), High confidence]
Type safety: Unchecked cast from Object to Map.Entry<String,ProgramPoint.LockState>
tests.NoUnlock$T3.run() does not release...l paths [Troubling(12), High confidence]

Excursion: Formal Verification

- Proving the correctness of an implementation with respect to a formal specification, using formal methods of mathematics.
- Formally prove that all possible executions of an implementation fulfill the specification

- Manual effort; partial automation; not automatically decidable

Testing

- Executing the program with selected inputs in a controlled environment
- Goals:
  – Reveal bugs (main goal)
  – Assess quality (hard to quantify)
  – Clarify the specification, documentation
  – Verify contracts

"Testing shows the presence,
not the absence of bugs

Edsger W. Dijkstra 1969

What to test?

- Functional correctness of a method (e.g., computations, contracts)
- Functional correctness of a class (e.g., class invariants)
- Behavior of a class in a subsystem/multiple subsystems/the entire system
- Behavior when interacting with the world
  - Interacting with files, networks, sensors, …
  - Erroneous states
  - Nondeterminism, Parallelism
  - Interaction with users
- Other qualities (performance, robustness, usability, security, …)

Our focus now

Manual Testing?

GENERIC TEST CASE: USER SENDS MMS WITH PICTURE ATTACHED.

Step ID	User Action	System Response
1	Go to Main Menu	Main Menu appears
2	Go to Messages Menu	Message Menu appears
3	Select “Create new Message”	Message Editor screen opens
4	Add Recipient	Recipient is added
5	Select “Insert Picture”	Insert Picture Menu opens
6	Select Picture	Picture is Selected
7	Select “Send Message”	Message is correctly sent

- Live System?
- Extra Testing System?
- Check output / assertions?
- Effort, Costs?
- Reproducible?

Automate Testing

- Execute a program with specific inputs, check output for expected values
- Easier to test small pieces than testing user interactions
- Set up testing infrastructure
- Execute tests regularly

Example

```
/**
 * computes the sum of the first len values of the array
 *
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the array values
 */
int total(int array[], int len);
```

Black box testing

Example

```
/**
 * computes the sum of the first len values of the array
 *
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the array values
 */
int total(int array[], int len);
```

- Test empty array
- Test array of length 1 and 2
- Test negative numbers
- Test invalid length (negative or longer than array.length)
- Test null as array
- Test with a very long array

Black box testing

Unit Tests

- Unit tests for small units: functions, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Little dependencies on other system parts or environment
- Insufficient but a good starting point, extra benefits:
  - Documentation (executable specification)
  - Design mechanism (design for testability)

JUnit

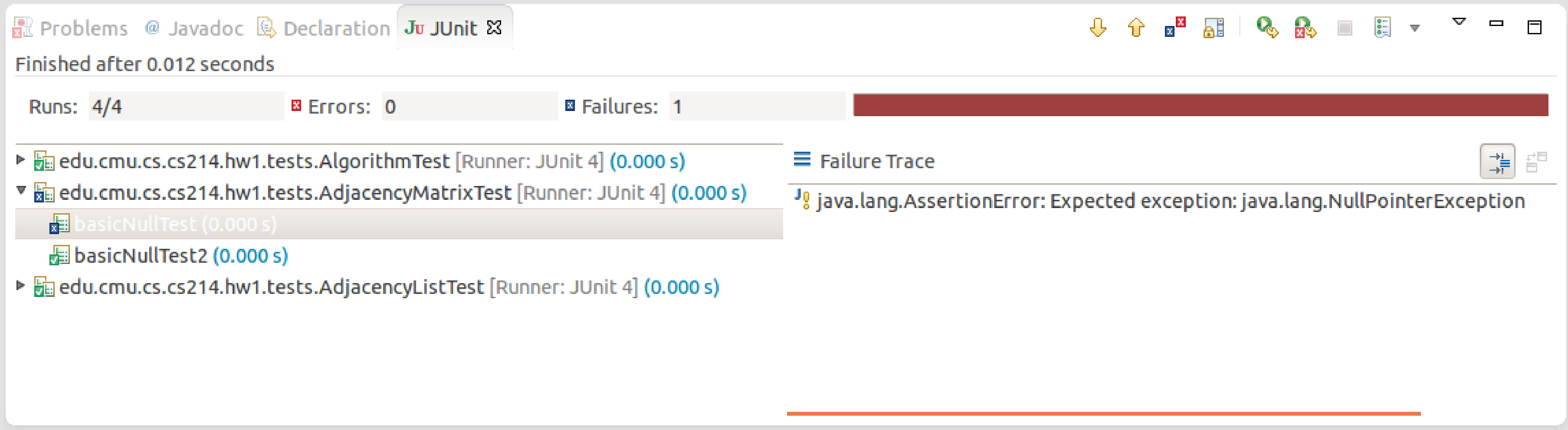
```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
      @Test
      public void testSanityTest(){
            Graph g1 = new AdjacencyListGraph(10);
            Vertex s1 = new Vertex("A");
            Vertex s2 = new Vertex("B");
            assertEquals(true, g1.addVertex(s1));
            assertEquals(true, g1.addVertex(s2));
            assertEquals(true, g1.addEdge(s1, s2));
            assertEquals(s2, g1.getNeighbors(s1)[0]);
      }

      @Test
      public void test….

      private int helperMethod…
}
```

Set up tests

Check expected results

JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available
- Can be used as design mechanism



institute for SOFTWARE RESEARCH

Selecting Test Cases: Common Strategies

- Read specification

- Write tests for representative case
  - Small instances are usually sufficient
- Write tests for invalid cases
- Write tests to check boundary conditions
- Are there difficult cases? (error guessing)
  - Stress tests? Complex algorithms?
- Think like an attacker
  - The tester's goal is to find bugs!

- Specification covered?
- Feel confident? Time/money left?

assert, Assert

- assert is a native Java statement throwing an AssertionError exception when failing
  - **assert** expression: "Error Message";
- org.junit.Assert is a library that provides many more specific methods
  - static void **assertTrue**(java.lang.String message, boolean condition)
    // Asserts that a condition is true.
  - static void **assertEquals**(java.lang.String message, long expected, long actual);
    // Asserts that two longs are equal.
  - static void **assertEquals**(double expected, double actual, double delta);
    // Asserts that two doubles are equal to within a positive delta
  - static void **assertNotNull**(java.lang.Object object)
    // Asserts that an object isn't null.
  - static void **fail**(java.lang.String message)
    //Fails a test with the given message.

JUnit Conventions

- TestCase collects multiple tests (in one class)
- TestSuite collects test cases (typically package)
- Tests should run fast
- Tests should be independent

- Tests are methods without parameter and return value
- AssertError signals failed test (unchecked exception)

- Test Runner knows how to run JUnit tests
  - (uses reflection to find all methods with @Test annotat.)

Common Setup

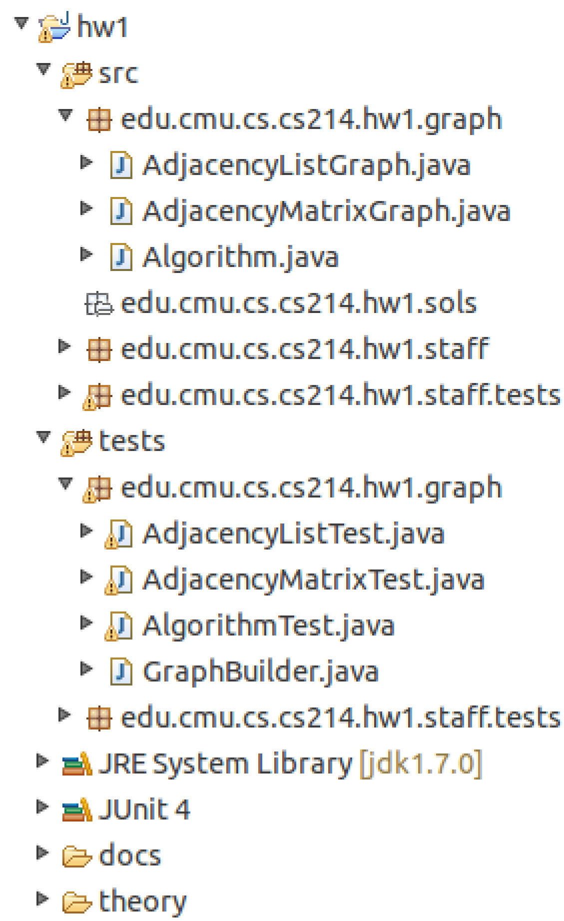
```
import org.junit.*;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
	Graph g;

	@Before
	public void setUp() throws Exception {
		graph = createTestGraph();
	}

	@Test
	public void testSanityTest(){
		Vertex s1 = new Vertex("A");
		Vertex s2 = new Vertex("B");
		assertEquals(3, g.getDistance(s1, s2));
	}
```

Checking for presence of an exception

```
import org.junit.*;
import static org.junit.Assert.fail;

public class Tests {

      @Test
      public void testSanityTest(){
            try {
                  openNonexistingFile();
                  fail("Expected exception");
            } catch(IOException e) { }
      }

      @Test(expected = IOException.class)
      public void testSanityTestAlternative() {
            openNonexistingFile();
      }
}
```

Test organization

- Conventions (not requirements)
- Have a test class XTest for each class X
- Have a source directory and a test directory
  – Store ATest and A in the same package
  – Tests can access members with default (package) visibility
  – Maven style: src/main/java and src/test/java



Testable Code

- Think about testing when writing code
- Unit testing encourages to write testable code
- Separate parts of the code to make them independently testable
- Abstract functionality behind interface, make it replaceable

- Test-Driven Development
  – A design and development method in which you write tests before you write the code!

Write testable code

```
//700LOC
public boolean foo() {
   try {
      synchronized () {
         if () {
         } else {
         }
         for () {
            if () {
               if () {
                  if () {
                     if ()?
                     {
                        if () {
                           for () {
                           }
                        }
                     }
                  } else {
                     if () {
                        for () {
                           if () {
                           } else {
                           }
                           if () {
                           } else {
                              if () {
                              }
                           }
                           if () {
                              if () {
                                 if () {
                                    for () {
                                    }
                                 }
                              }
                           } else {
                           }
                        }
```

Unit testing as design mechanism

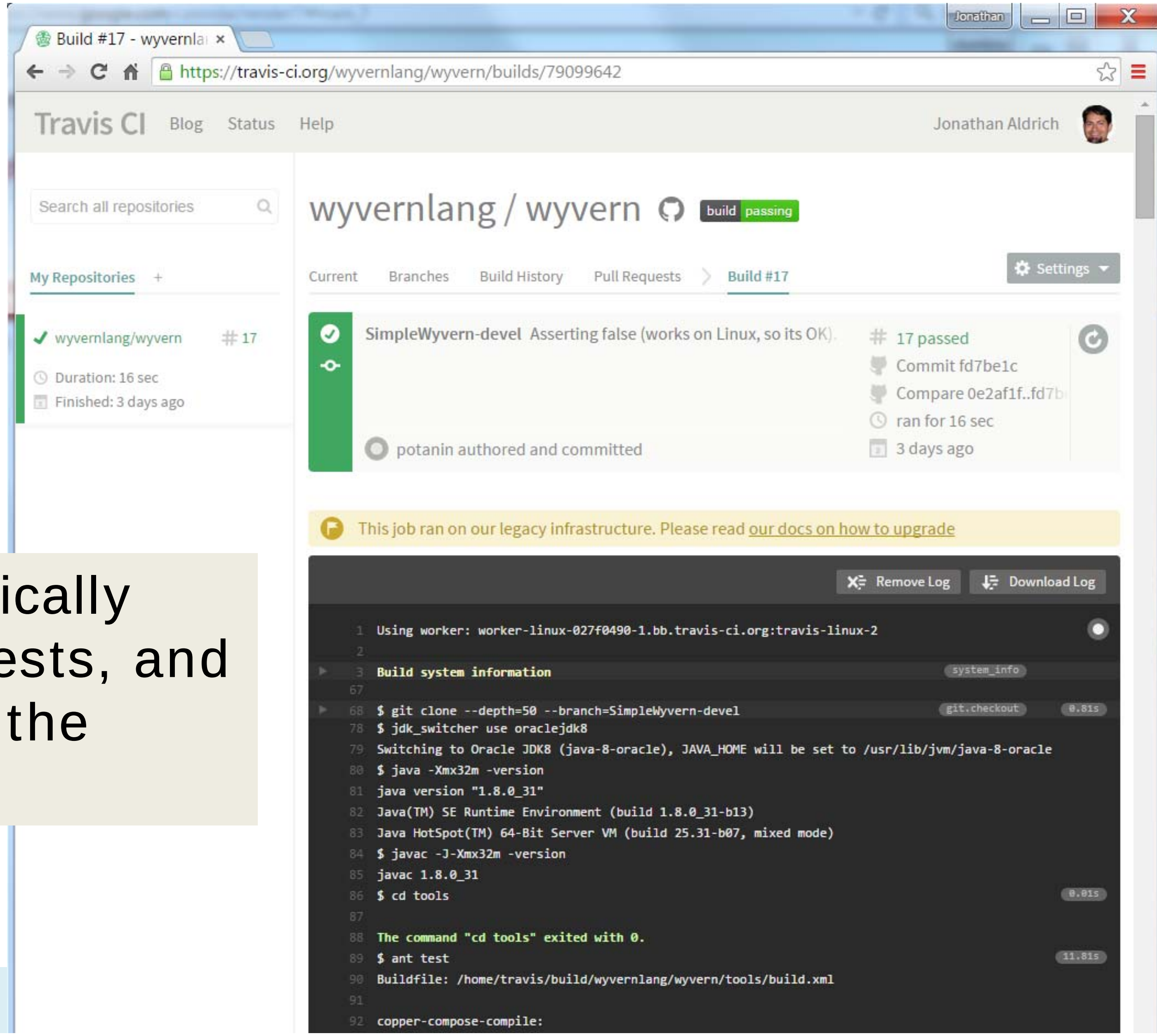
- **Code with low complexity**
- **Clear interfaces and specifications**

Source: http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx

Run tests frequently

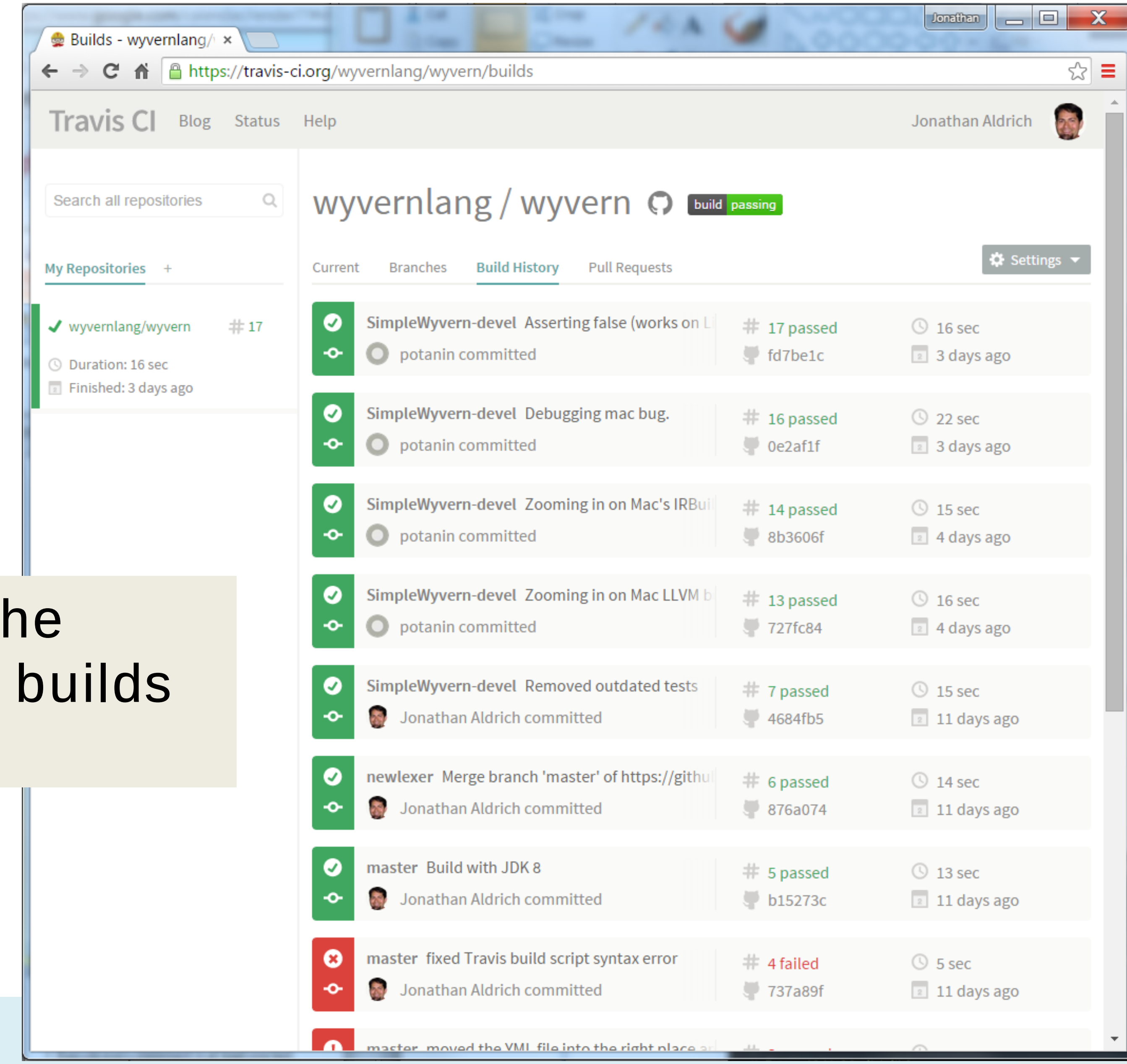
- You should only commit code that is passing all tests
- Run tests before every commit
- Run tests before trying to understand other developers' code
- If entire test suite becomes too large and slow for rapid feedback, run local tests ("smoke tests", e.g. all tests in package) frequently, run all tests nightly
  - Medium sized projects easily have 1000s of test cases and run for minutes
- Continuous integration servers help to scale testing

Continuous Integration - Travis CI



Automatically builds, tests, and displays the result

Continuous Integration - Travis CI



Can see the results of builds over time

Outlook: Statement Coverage

- Trying to test all parts of the implementation
- Execute every statement in at least one test

```java
    }
    public boolean equals(Object anObject) {
        if (isZero())
            if (anObject instanceof IMoney)
                return ((IMoney)anObject).isZero();
        if (anObject instanceof Money) {
            Money aMoney= (Money)anObject;
            return aMoney.currency().equals(currency())
                             && amount() == aMoney.amount();
        }
        return false;
    }
    public int hashCode() {
```

- Does this guarantee correctness?

institute for SOFTWARE RESEARCH

Sample.java | coverage-test/pom.xml | coverage-test 0.0.1-... | SampleTest.java

```java
    }

    public int subtract(int a, int b) {
        int x = a - b;

        return x;
    }

    public boolean conditional(int a, int b) {
        return a == b;
    }

    public void uncoveredMethod() {
        String line = "not covered";
    }

    public String coveredMethod() {
        String a = "hello"; String b = "world"; return a.concat(b);
    }
}
```

Problems | Javadoc | Declaration | Console | Search | Coverage | Coverage Sessi | Clover Dashbo | Coverage Expl | Test Run Explo | Test Cont

Name	Lines	Total	%	Branches	Total
All Packages (2010-10-21 21:38:34)	38	46	82.61 %	1	18
com.copperykeenclaws	38	46	82.61 %	1	18
Sample	11	14	78.57 %	1	2
Sample$__CLR3_0_100gfkf1nq8	1	1	100.00 %	0	0
SampleTest	25	30	83.33 %	0	16

Packages

All
net.sourceforge.cobertura.ant
net.sourceforge.cobertura.check
net.sourceforge.cobertura.coveragedat
net.sourceforge.cobertura.instrument
net.sourceforge.cobertura.merge
net.sourceforge.cobertura.reporting
net.sourceforge.cobertura.reporting.htr
net.sourceforge.cobertura.reporting.htr
net.sourceforge.cobertura.reporting.xm
net.sourceforge.cobertura.util

All Packages

Classes

AntUtil *(88%)*
Archive *(100%)*
ArchiveUtil *(80%)*
BranchCoverageData *(N/A)*
CheckTask *(0%)*
ClassData *(N/A)*
ClassInstrumenter *(94%)*
ClassPattern *(100%)*
CoberturaFile *(73%)*
CommandLineBuilder *(96%)*
CommonMatchingTask *(88%)*
ComplexityCalculator *(100%)*
ConfigurationUtil *(50%)*
CopyFiles *(87%)*
CoverageData *(N/A)*
CoverageDataContainer *(N/A)*
CoverageDataFileHandler *(N/A)*
CoverageRate *(0%)*
ExcludeClasses *(100%)*
FileFinder *(96%)*
FileLocker *(0%)*
FirstPassMethodInstrumenter *(100%)*
HTMLReport *(94%)*
HasBeenInstrumented *(N/A)*
Header *(80%)*

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Compl
All Packages	55	75%	1625/2179	64%	472/738	
net.sourceforge.cobertura.ant	11	52%	170/330	43%	40/94	
net.sourceforge.cobertura.check	3	0%	0/150	0%	0/76	
net.sourceforge.cobertura.coveragedata	13	N/A	N/A	N/A	N/A	
net.sourceforge.cobertura.instrument	10	90%	460/510	75%	123/164	
net.sourceforge.cobertura.merge	1	86%	30/35	88%	14/16	
net.sourceforge.cobertura.reporting	3	87%	116/134	80%	43/54	
net.sourceforge.cobertura.reporting.html	4	91%	475/523	77%	156/202	
net.sourceforge.cobertura.reporting.html.files	1	87%	39/45	62%	5/8	
net.sourceforge.cobertura.reporting.xml	1	100%	155/155	95%	21/22	
net.sourceforge.cobertura.util	9	60%	175/291	69%	70/102	
someotherpackage	1	83%	5/6	N/A	N/A	

Report generated by Cobertura 1.9 on 6/9/07 12:37 AM.

Testing, Static Analysis, and Proofs

- Testing
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now
  - Does not find all problems (unsound)

- Static Analysis
  - Analysis of all possible executions
  - Specific issues only with conservative approx. and bug patterns
  - Tools available, useful for bug finding
  - Automated, but unsound and/or incomplete

- Proofs (Formal Verification)
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - Practical for small programs, may scale up in the future
  - Sound and complete, but not automatically decidable

What strategy to use in your project?

DESIGN GUIDELINES

Avoid Global State

- Bad
  - Module A writes data to global (static) variable X
  - Module B reads from X
  - Why? Hard to specify, understand, and change
- Good
  - Module A creates an object with data
  - Module A calls B passing the data object

Avoid static, instanceof, and casts

- Bad

```
static void foo(A x) {
   if (x instanceof B) {
         B b = (B) x;
         // handle B's
   } else if (x instanceof C) {
         // handle C's
   }
}

// in main
foo(anA);
```

- Good

```
interface A {
   void foo();
}
class B extends A {
   void foo() {
         // handle B's
   }
}

// in main
anA.foo()
```

The OO version makes it easier to
- Understand each class in isolation
- Add new classes later

Java: Static Methods

- Static methods belong to a **class**, not an object
- They are global (a single implementation only)
- Direct dispatch, no subtype polymorphism
- Avoid unless really only a single implementation exists (e.g., Math.min)
- Pure object-oriented languages don't support static methods

```
Point p = …
p.getX()

Point.move(p);
```


119

Java: Breaking encapsulation: instanceof and typecast

- Java allows to inspect an object's runtime type

```
Point p = ...
if (p instanceof PolarPoint) {
    PolarPoint q = (PolarPoint) p;
    q.getAngle()
}
```

- Objects always assignable to variables of supertypes ("upcast")

```
CartesianPoint q = ...
Point p = q;
```

(this effectively throws away parts of the interface)

- Assignment to subtype requires downcast (may fail at runtime!)

```
Point p = ...
CartesianPoint q = (CartesianPoint) p;
```

Avoid instanceof and downcasts

Instanceof breaks encapsulation

- Never ask for the type of an object
- Instead, ask the object to do something (call a method of the interface)
- If the interface does not provide the method, maybe there was a reason? Rethink design!

- Instanceof and downcasts are indicators of poor design
- They break abstractions and encapsulation
- There are only few exceptions where **instanceof** is needed
- Use polymorphism instead

- Pure object-oriented languages do not have an instanceof operation

Exercise: instanceof

```
void test() {
    Expr e = new Add(new Lit(1), new Minus(new Lit(2), new Lit(0)));
    System.out.println(evaluate(e));
    System.out.println(print(e));
}
interface Expr { }
class Lit implements Expr {
    int value;   Lit(int a) { this.value = a;}
}
class Add implements Expr {
    Expr a, b;   Add(Expr x, Expr y) { this.a = x; this.b = y; }
}
class Minus implements Expr {
    Expr a, b;   Minus(Expr x, Expr y) { this.a = x; this.b = y; }
}
int evaluate(Expr e) {
    if (e instanceof Lit) return ((Lit) e).value;
    if (e instanceof Add) return evaluate(((Add)e).a) + evaluate(((Add)e).b);
    if (e instanceof Minus) return evaluate(((Minus)e).a) - evaluate(((Minus)e).b);
    return 0;
}
String print(Expr e) {
    if (e instanceof Lit) return Integer.toString(((Lit) e).value);
    if (e instanceof Add) return "(" + print(((Add)e).a) + " + " + print(((Add)e).b) + ")";
    if (e instanceof Minus) return "(" + print(((Minus)e).a) + " - " + print(((Minus)e).b) + ")";
    return "";
}
```

1. Rewrite the code to not use instanceof.

2. In each implementation: Add *Power* expression and *printHex* function

EXCURSION: TECHNICAL REALIZATION OF SUBTYPE POLYMORPHISM

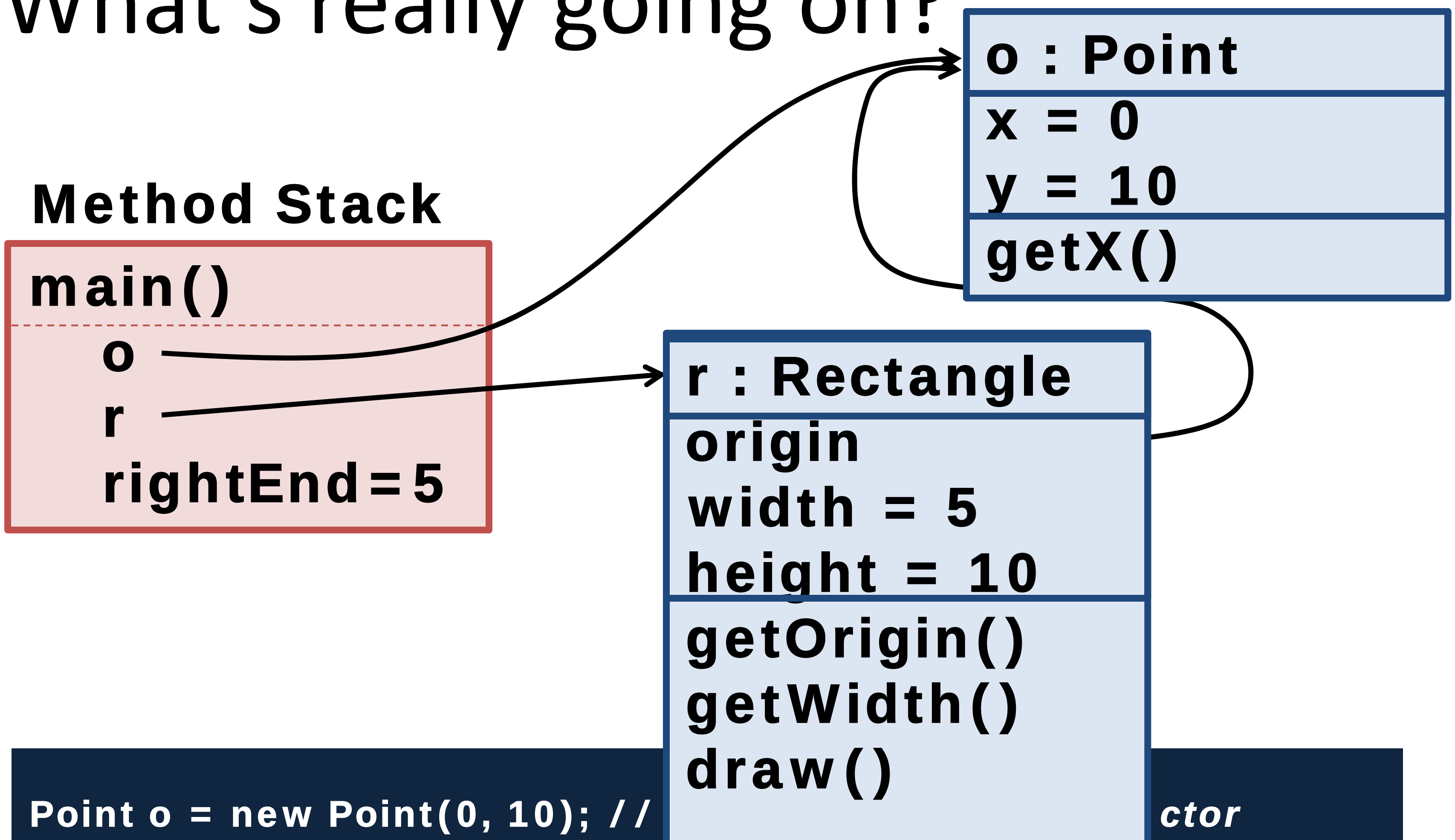
Reminder: Subtype Polymorphism

- A type (e.g. Point) can have many forms (e.g., CartesianPoint, PolarPoint, …)
- All implementations of an interface can be used interchangeably
- When invoking a method p.x() the specific implementation of x() from object p is executed
  - The executed method depends on the actual object p, i.e., on the runtime type
  - It does not depend on the static type, i.e., how p is declared

Objects and References (example)

```
// allocates memory, calls constructor
Point o = new PolarPoint(0, 10);

Rectangle r = new MyRectangle(o, 5, 10);

r.draw();

int rightEnd = r.getOrigin().getX() +
    r.getWidth(); // 5
```

What's really going on?




```
Point o = new Point(0, 10); // … ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5
```

Anatomy of a Method Call

`r.setX(5)`

The **receiver**,
an implicit argument,
called **this** inside the
method

Method **arguments**,
just like function
arguments

The method **name**.
Identifies which method to use,
of all the methods the receiver's
class defines

Java Specifics: The keyword **this** refers to the "receiver"

```
class Point {
    int x, y;
    int getX() { return this.x; }
    Point(int x, int y) { this.x = x; this.y = y; }
}
```

can also be written in this way:

```
class Point {
    int x, y;
    int getX() { return x; }
    Point(int px, int py) { x = px; y = py; }
}
```

Static types vs dynamic types

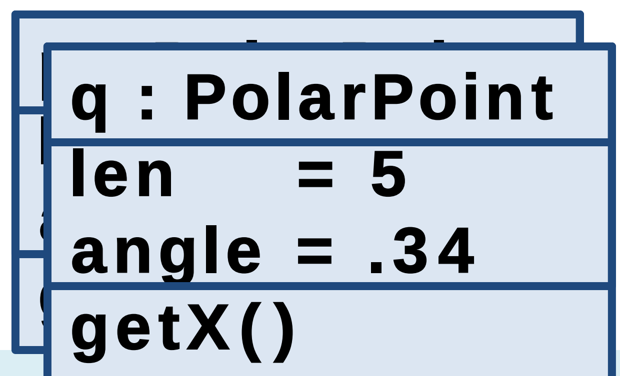
- Static type: how is a variable declared
- Dynamic type: what type has the object in memory when executing the program (we may not know until we execute the program)

```
Point createZeroPoint() {
        if (new Math.Random().nextBoolean())
                return new CartesianPoint(0, 0);
        else    return new PolarPoint(0,0);
}
Point p = createZeroPoint();
p.getX();
p.getAngle();
```

Method dispatch (conceptually)

- Step 1 (compile time): determine what type to look in
  - Look at the **static type** (Point) of the receiver (p)
- Step 2 (compile time): find the method in that type
  - Find the method in the **interface/class** with the right name

    int getX();
  - Error if there is no such method
  - Error if the method is not accessible (e.g., private)
- Step 3 (run time): Execute the method stored in the **object**



Method dispatch (actual; simplified)

- Step 3 (run time): Determine the run-time type of the receiver
  - Look at the object in the **heap** and get its **class**
- Step 4 (run time): Locate the method implementation to invoke
  - Look in the class for an implementation of the method
  - Invoke that implementation

SUMMARY: DESIGN FOR CHANGE/ DIVISION OF LABOR

Design Goals

- Design for Change such that
  – Classes are *open for extension* and modification without invasive changes
  – Subtype polymorphism enables changes behind interface
  – Classes encapsulate details likely to change behind (small) stable interfaces
- Design for Division of Labor such that
  – Internal parts can be *developed* independently
  – Internal details of other classes do not need to be *understood*, contract is sufficient
  – Test classes and their contracts separately (unit testing)

Aside: UML class diagram notation