

Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing Classes

UML + Design Patterns

Charlie Garrod

Michael Hilton

Administrivia

- HW2 due Thursday Sept 14, 11:59 pm

Readings

- Optional reading due today:
 - Item 16: Favor composition over inheritance
 - Item 17: Design and document for inheritance or else prohibit it
 - Item 18: Prefer interfaces to abstract classes
- Tuesday required readings due:
 - Chapter 9. Use-Case Model: Drawing System Sequence Diagrams
 - Chapter 10. Domain Model: Visualizing Concepts

Plan for today

- UML
- Intro to design patterns

Diagrams

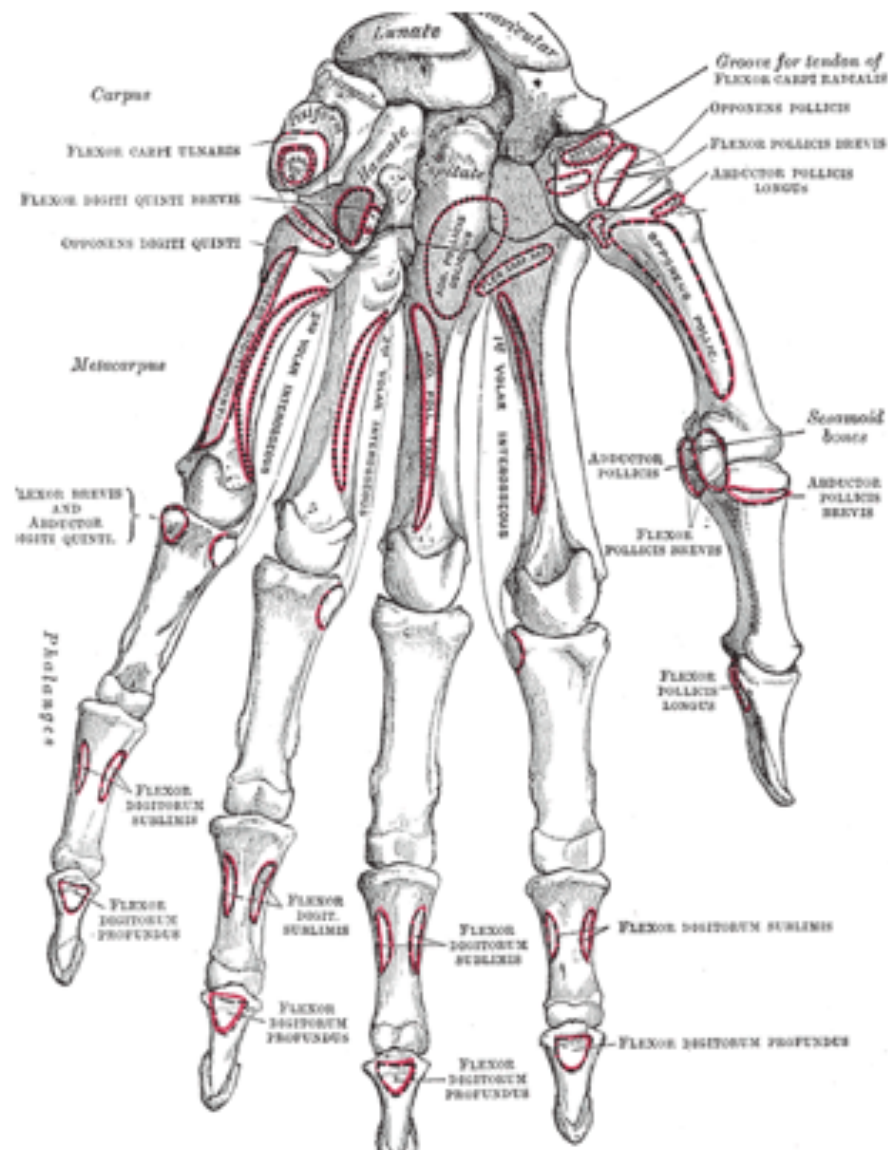
Are often useful when you need to:

Communicate,

Visualize or

Analyze

something, especially something with some structure.



https://en.wikipedia.org/wiki/Gray%27s_Anatomy#/media/File:Gray219.png

Mr. James Wilson, M.Inst.C.E., Engineer.

SIGNALLING PLAN OF LIVERPOOL STREET STATION
GREAT EASTERN RAILWAY.
 MR. JOHN WILSON, M.Inst.C.E., Engineer.

Points in use on East Side for 10 Points
 (See Notes)
 10 11 12 13 14 15 16 17 18 19
 20 21 22 23 24 25 26 27 28 29
 30 31 32 33 34 35 36 37 38 39

Points in use on West Side for 10 Points
 (See Notes)
 40 41 42 43 44 45 46 47 48 49
 50 51 52 53 54 55 56 57 58 59
 60 61 62 63 64 65 66 67 68 69

Points in use on East Side for 10 Points
 (See Notes)
 70 71 72 73 74 75 76 77 78 79
 80 81 82 83 84 85 86 87 88 89
 90 91 92 93 94 95 96 97 98 99

Points in use on West Side for 10 Points
 (See Notes)
 100 101 102 103 104 105 106 107 108 109
 110 111 112 113 114 115 116 117 118 119
 120 121 122 123 124 125 126 127 128 129

Points in use on East Side for 10 Points
 (See Notes)
 130 131 132 133 134 135 136 137 138 139
 140 141 142 143 144 145 146 147 148 149
 150 151 152 153 154 155 156 157 158 159

Points in use on West Side for 10 Points
 (See Notes)
 160 161 162 163 164 165 166 167 168 169
 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189

Points in use on East Side for 10 Points
 (See Notes)
 190 191 192 193 194 195 196 197 198 199
 200 201 202 203 204 205 206 207 208 209
 210 211 212 213 214 215 216 217 218 219

Points in use on West Side for 10 Points
 (See Notes)
 220 221 222 223 224 225 226 227 228 229
 230 231 232 233 234 235 236 237 238 239
 240 241 242 243 244 245 246 247 248 249

Points in use on East Side for 10 Points
 (See Notes)
 250 251 252 253 254 255 256 257 258 259
 260 261 262 263 264 265 266 267 268 269
 270 271 272 273 274 275 276 277 278 279

Points in use on West Side for 10 Points
 (See Notes)
 280 281 282 283 284 285 286 287 288 289
 290 291 292 293 294 295 296 297 298 299
 300 301 302 303 304 305 306 307 308 309

Points in use on East Side for 10 Points
 (See Notes)
 310 311 312 313 314 315 316 317 318 319
 320 321 322 323 324 325 326 327 328 329
 330 331 332 333 334 335 336 337 338 339

Points in use on West Side for 10 Points
 (See Notes)
 340 341 342 343 344 345 346 347 348 349
 350 351 352 353 354 355 356 357 358 359
 360 361 362 363 364 365 366 367 368 369

Points in use on East Side for 10 Points
 (See Notes)
 370 371 372 373 374 375 376 377 378 379
 380 381 382 383 384 385 386 387 388 389
 390 391 392 393 394 395 396 397 398 399

Points in use on West Side for 10 Points
 (See Notes)
 400 401 402 403 404 405 406 407 408 409
 410 411 412 413 414 415 416 417 418 419
 420 421 422 423 424 425 426 427 428 429

Points in use on East Side for 10 Points
 (See Notes)
 430 431 432 433 434 435 436 437 438 439
 440 441 442 443 444 445 446 447 448 449
 450 451 452 453 454 455 456 457 458 459

Points in use on West Side for 10 Points
 (See Notes)
 460 461 462 463 464 465 466 467 468 469
 470 471 472 473 474 475 476 477 478 479
 480 481 482 483 484 485 486 487 488 489

Points in use on East Side for 10 Points
 (See Notes)
 490 491 492 493 494 495 496 497 498 499
 500 501 502 503 504 505 506 507 508 509
 510 511 512 513 514 515 516 517 518 519

Points in use on West Side for 10 Points
 (See Notes)
 520 521 522 523 524 525 526 527 528 529
 530 531 532 533 534 535 536 537 538 539
 540 541 542 543 544 545 546 547 548 549

Points in use on East Side for 10 Points
 (See Notes)
 550 551 552 553 554 555 556 557 558 559
 560 561 562 563 564 565 566 567 568 569
 570 571 572 573 574 575 576 577 578 579

Points in use on West Side for 10 Points
 (See Notes)
 580 581 582 583 584 585 586 587 588 589
 590 591 592 593 594 595 596 597 598 599
 600 601 602 603 604 605 606 607 608 609

Points in use on East Side for 10 Points
 (See Notes)
 610 611 612 613 614 615 616 617 618 619
 620 621 622 623 624 625 626 627 628 629
 630 631 632 633 634 635 636 637 638 639

Points in use on West Side for 10 Points
 (See Notes)
 640 641 642 643 644 645 646 647 648 649
 650 651 652 653 654 655 656 657 658 659
 660 661 662 663 664 665 666 667 668 669

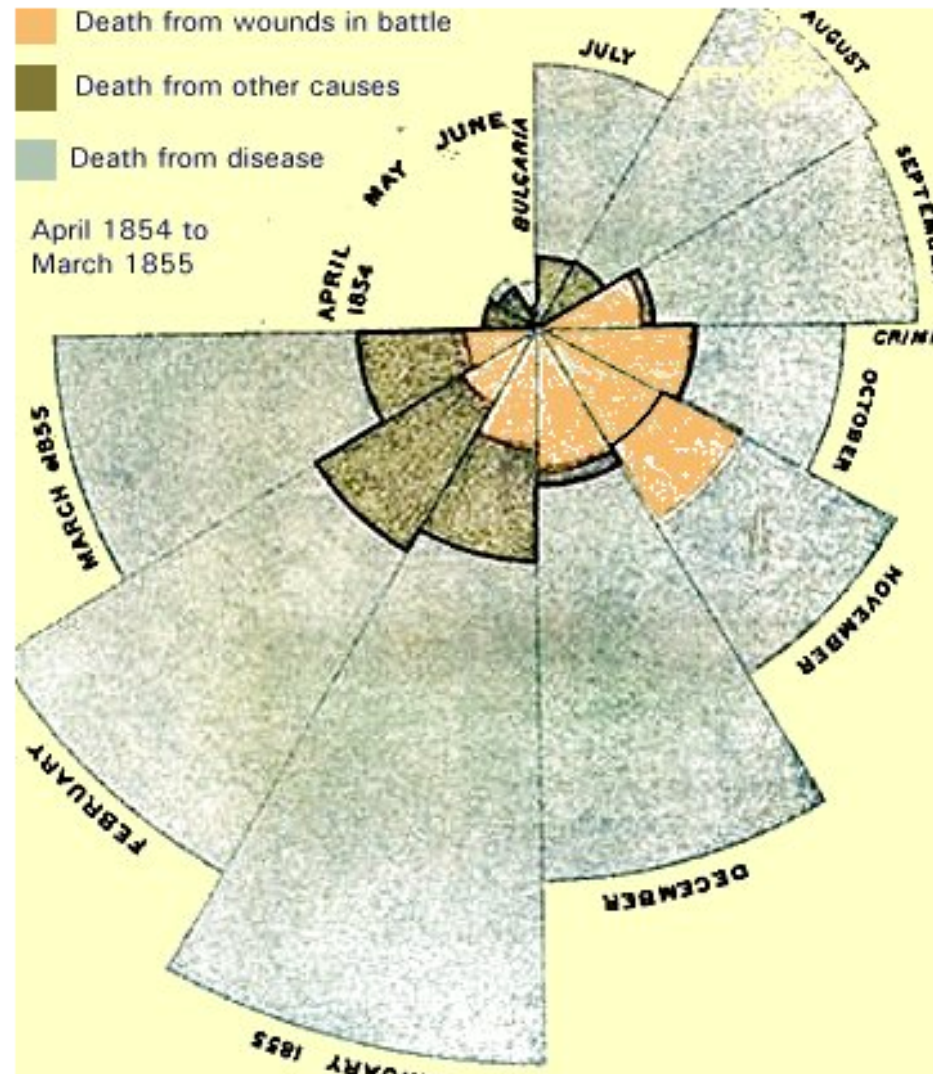
Points in use on East Side for 10 Points
 (See Notes)
 670 671 672 673 674 675 676 677 678 679
 680 681 682 683 684 685 686 687 688 689
 690 691 692 693 694 695 696 697 698 699

Points in use on West Side for 10 Points
 (See Notes)
 700 701 702 703 704 705 706 707 708 709
 710 711 712 713 714 715 716 717 718 719
 720 721 722 723 724 725 726 727 728 729

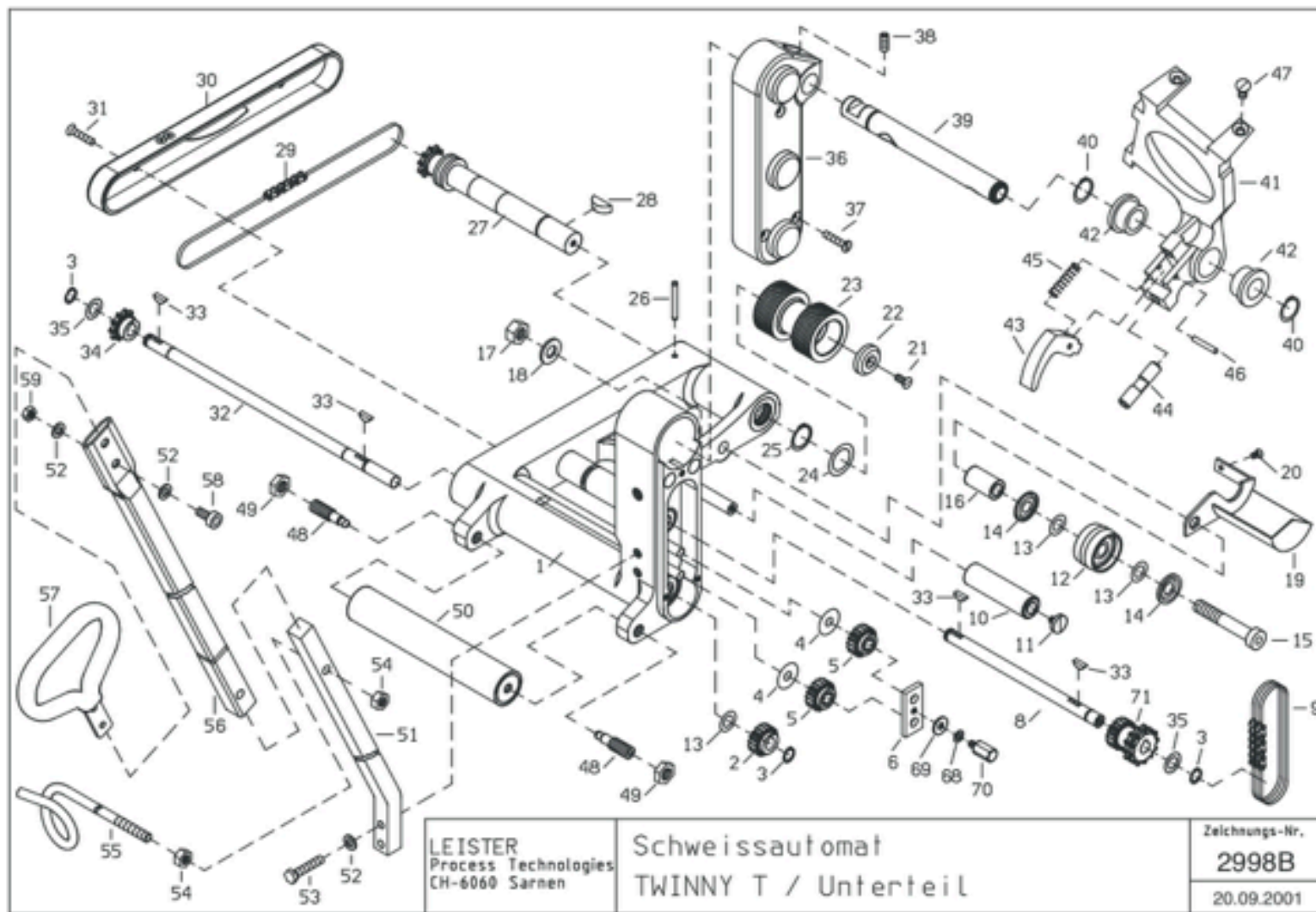
Points in use on East Side for 10 Points
 (See Notes)
 730 731 732 733 734 735 736 737 738 739
 740 741 742 743 744 745 746 747 748 749
 750 751 752 753 754 755 756 757 758 759

Points in use on West Side for 10 Points
 (See Notes)
 760 761 762 763 764 765 766 767 768 769
 770 771 772 7

15-214



<http://www.uh.edu/engines/epi1712.htm>

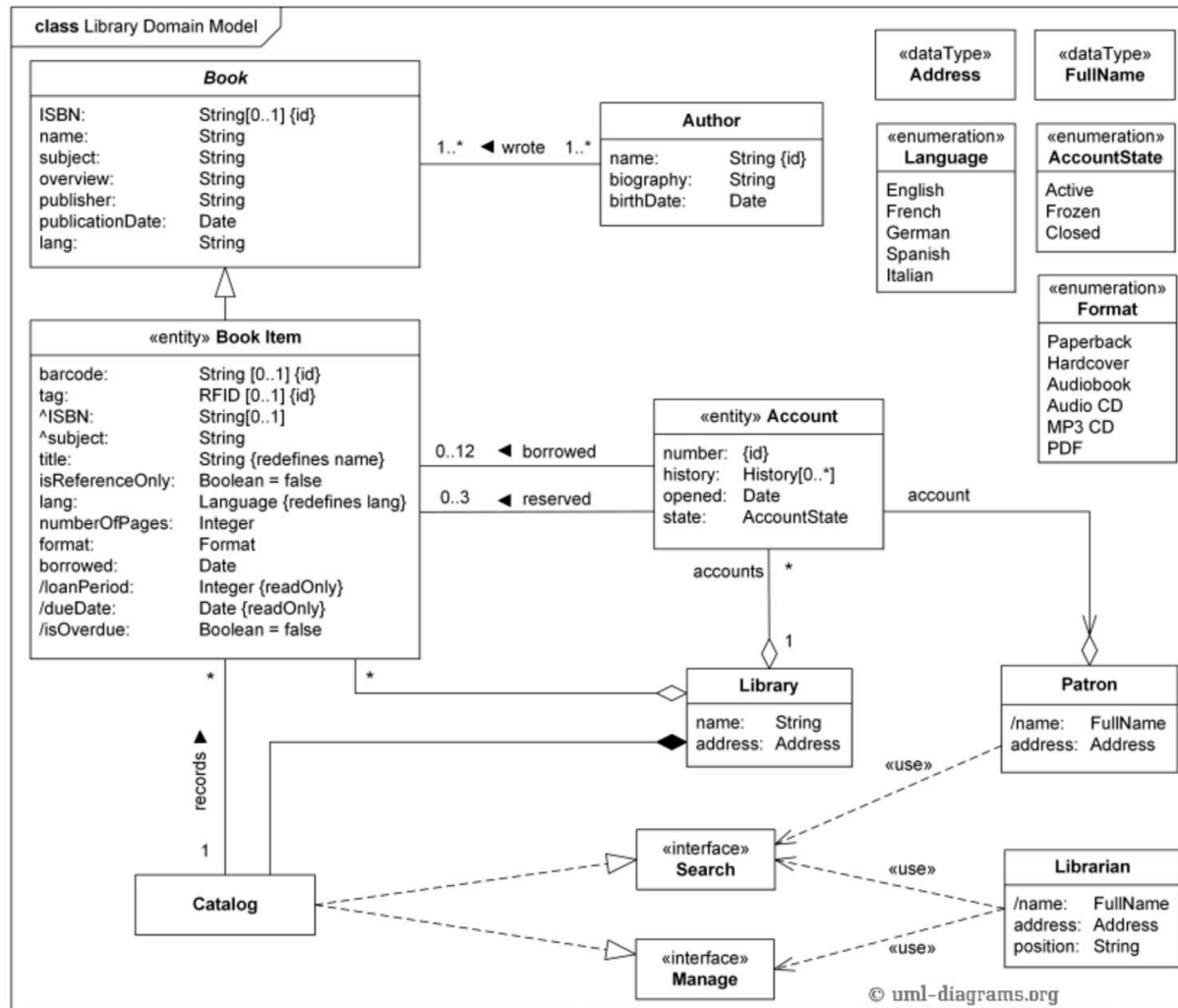


http://files.www.ihshotair.com/twinny-s/0508-Machine_Base-CE_version.jpg

***“Democracy is the
worst form of
government, except for
all the others”***

-Allegedly Winston Churchill

UML: Unified Modeling Language



UML in this course

- UML class diagrams
- UML interaction diagrams
 - Sequence diagrams
 - Communication diagrams

UML class diagrams (interfaces and inheritance)

```
public interface Account {  
    public long getBalance();  
    public void deposit(long amount);  
    public boolean withdraw(long amount);  
    public boolean transfer(long amount, Account target);  
    public void monthlyAdjustment();  
}
```

```
public interface CheckingAccount extends Account {  
    public long getFee();  
}
```

```
public interface SavingsAccount extends Account {  
    public double getInterestRate();  
}
```

```
public interface InterestCheckingAccount  
    extends CheckingAccount, SavingsAccount {  
}
```


UML class diagrams (classes)

```
public abstract class AbstractAccount
    implements Account {
    protected long balance = 0;
    public long getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}
```

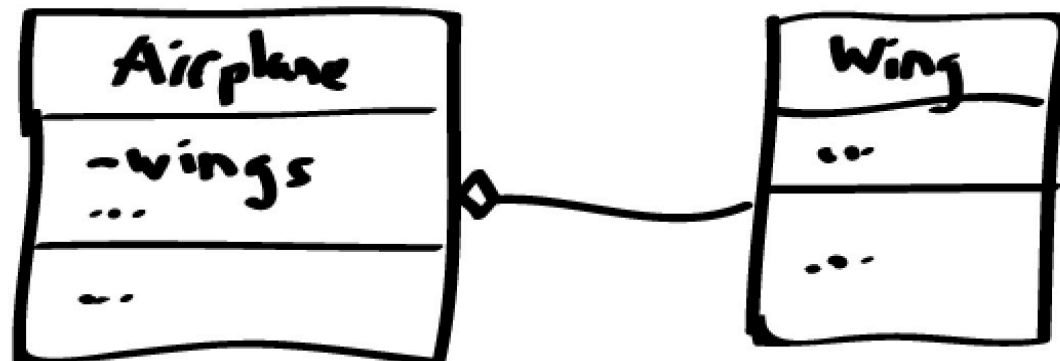
```
public class CheckingAccountImpl
    extends AbstractAccount
    implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public long getFee() { ... }
}
```


UML you should know

- Interfaces vs. classes
- Fields vs. methods
- Relationships:
 - "extends" (inheritance)
 - "implements" (realization)
 - "has a" (aggregation)
 - non-specific association
- Visibility: + (public) - (private) # (protected)
- Basic best practices...

UML advice

- Best used to show the big picture
 - Omit unimportant details
 - But show they are there: ...
- Avoid redundancy
 - e.g., bad:



good:



```

public class Processor {
    ClaimApproval ca;
    FloodClaimValidator fcv = new
FloodClaimValidator();
    FireClaimValidator ficv = new
FireClaimValidator();

    public void setupClaims(){...}

    public boolean processClaims(){
        ca = new ClaimApproval();
        if(ca.processClaim(fcv) &&
ca.processClaim(ficv)){
            return true;
        }
        else return false;
    }
}

```

```

public class ClaimApproval {
    public boolean processClaim(AbstractValidator
validator){
        if(validator.isClaimValid()){
            System.out.println("Claim is approved");
            return true;
        }
        return false;
    }
}

public abstract class AbstractValidator {
    public abstract boolean isClaimValid();
}

-----
-----

public class FireClaimValidator extends
AbstractValidator {
    public boolean isClaimValid(){
        System.out.println("valid fire claim");
        return true;
    }
}

-----
-----

public class FloodClaimValidator extends
AbstractValidator {
    public boolean isClaimValid(){
        System.out.println("validating claim");
        return true;
    }
}

```

One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

A third design scenario

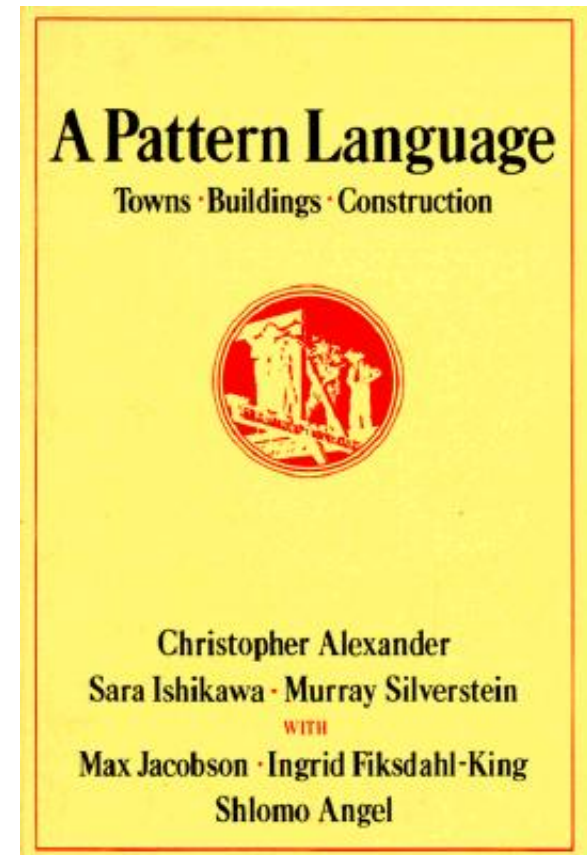
- Suppose we need to sort a list in different orders...

```
interface Comparator {  
    boolean compare(int i, int j);  
}  
  
final Comparator ASCENDING = (i, j) -> i < j;  
final Comparator DESCENDING = (i, j) -> i > j;  
  
static void sort(int[] list, Comparator cmp) {  
    ...  
    boolean mustSwap =  
        cmp.compare(list[i], list[j]);  
    ...  
}
```

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,
Architect (1977)



DESIGN PATTERNS

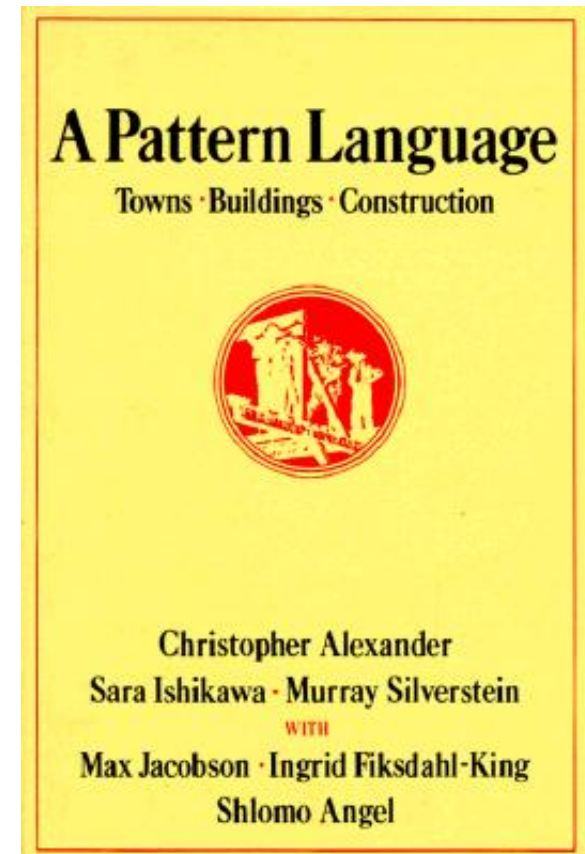
Christopher Alexander



- By Michaelmehaffy - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=47871494>

Christopher Alexander

- Worked as in computer science but trained as an architect
- Wrote a book called A Pattern Language: Towns, Buildings, Construction.



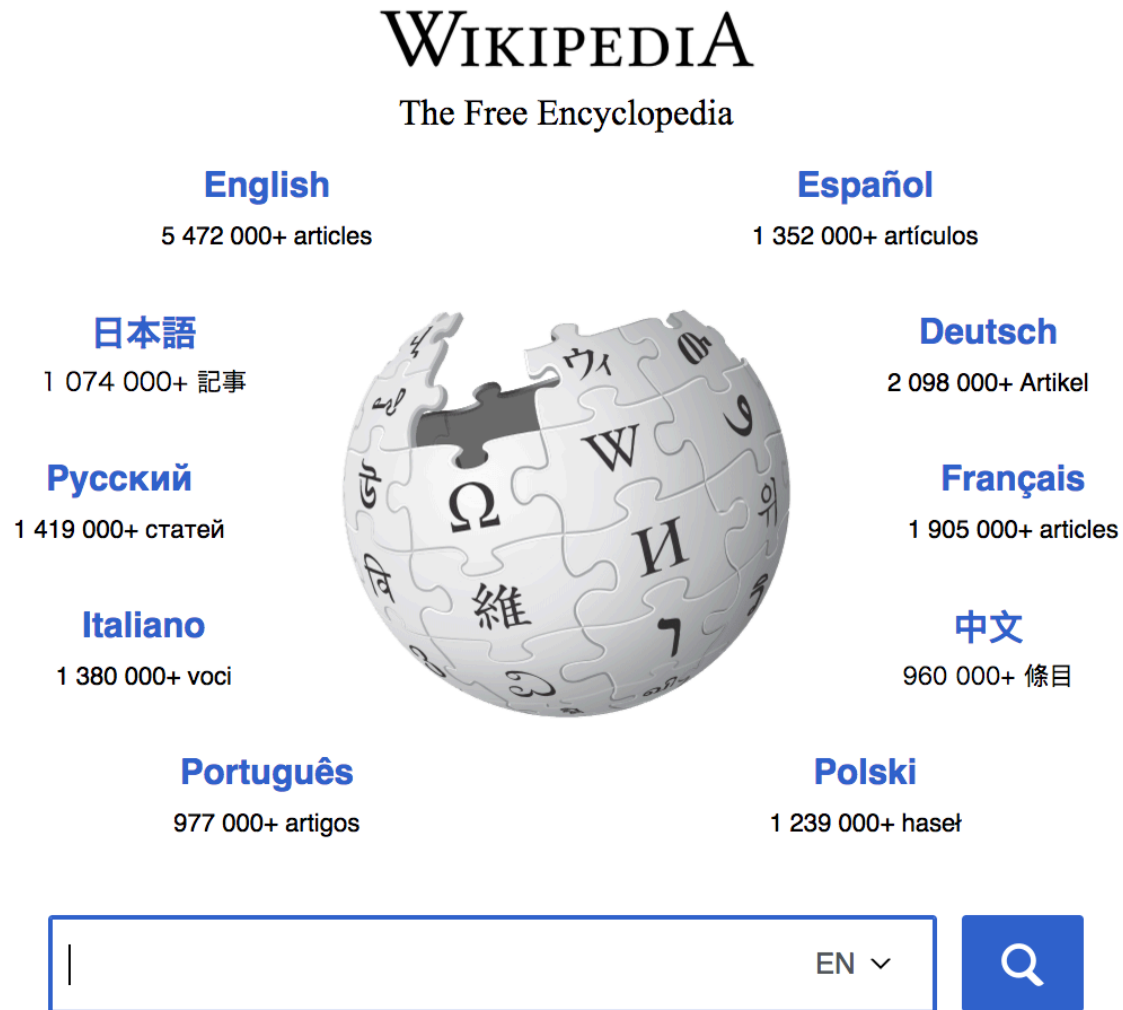
The timeless Way of Building

- Asks the question, “is quality objective?”
- Specifically, “What makes us know when an architectural design is good? Is there an objective basis for such a judgement?”
- He studied the problem of identifying what makes a good architectural design by observing:
 - Buildings
 - Towns
 - Streets
 - homes
 - community centers
 - etc.
- When he found a good example, he would compare with others.

Four Elements of a Pattern

- Alexander identified four elements to describe a pattern:
 - The name of the pattern
 - The purpose of the pattern: what problem it solves
 - How to solve the problem
 - The constraints we have to consider in our solution

Inspired by Alexanders Work



Inspired by Alexanders Work



Inspired by Alexanders Work



Software design patterns

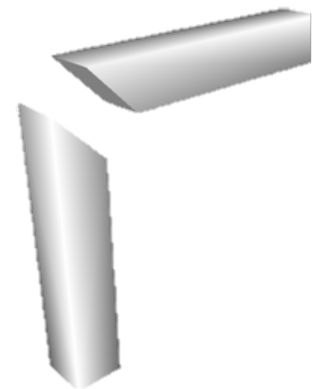
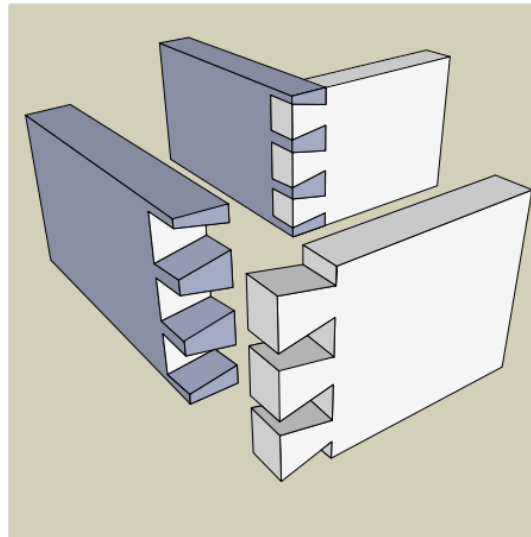
- Are there problems in software that occur all the time that can be solved in somewhat the same manner?
- Is it possible to design software in terms of patterns?

How not to discuss design (from Shalloway and Trott)

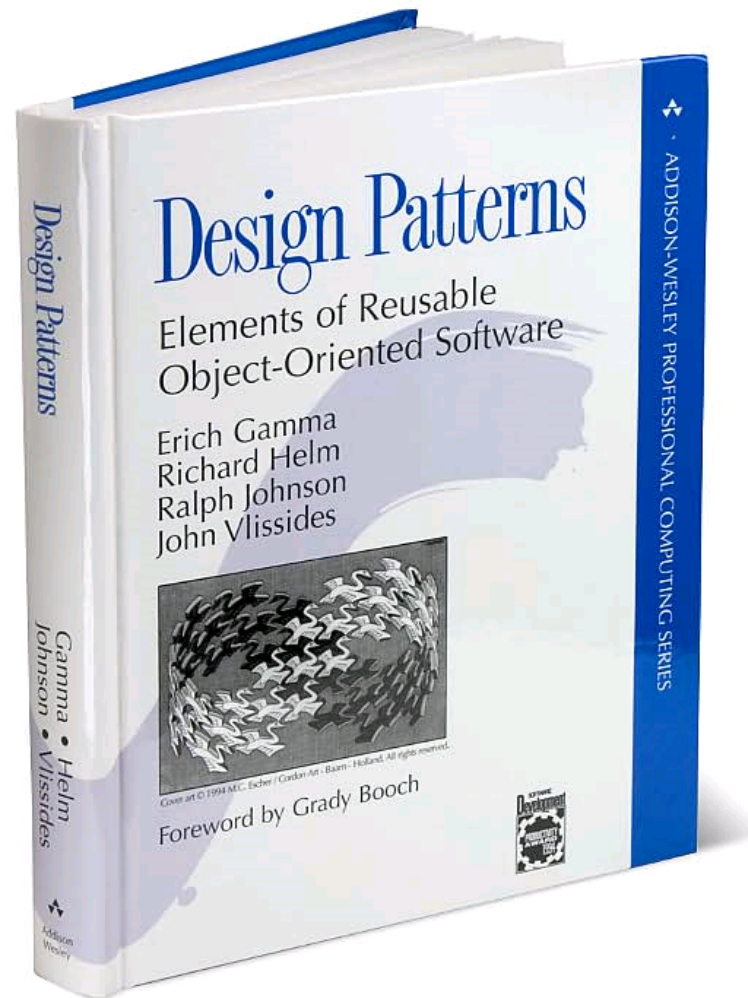
- Carpentry:
 - How do you think we should build these drawers?
 - Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating...
- Software Engineering:
 - How do you think we should write this method?
 - I think we should write this if statement to handle ... followed by a while loop ... with a break statement so that...

Discussion with design patterns

- Carpentry:
 - "Is a dovetail joint or a miter joint better here?"
- Software Engineering:
 - "Is a strategy pattern or a template method better here?"



History: *Design Patterns* (1994)



Elements of a design pattern

- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences

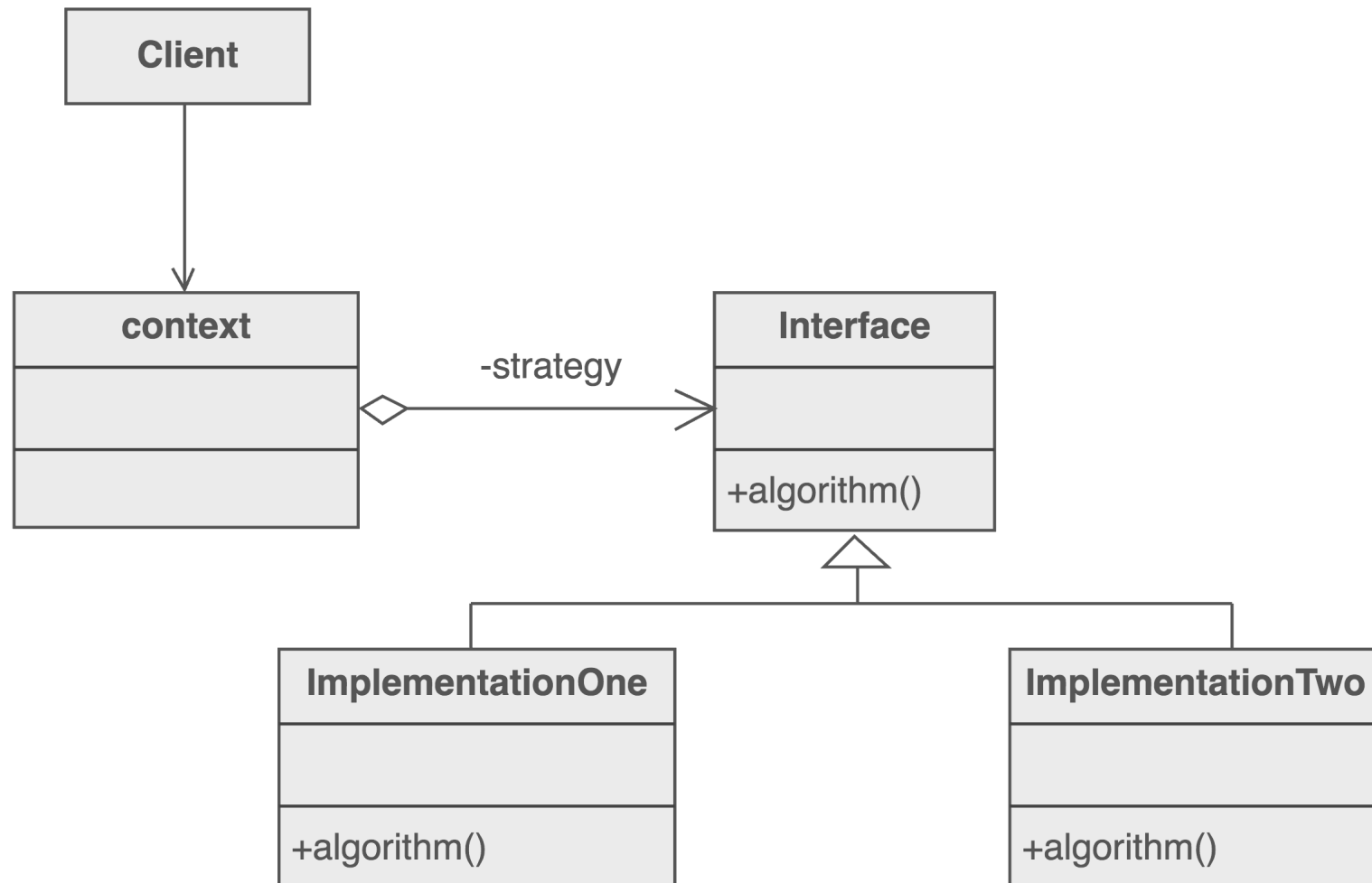
Recognizing a pattern

- Amazon tax
- Computer Vision
- List Sorting

Strategy pattern

- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
 - Easily extensible for new algorithm implementations
 - Separates algorithm from client context
 - Introduces an extra interface and many classes:
 - Code can be harder to understand
 - Lots of overhead if the strategies are simple

Strategy Pattern - UML



https://sourcemaking.com/design_patterns/strategy

Patterns are more than just structure

- Consider: A modern car engine is constantly monitored by a software system. The monitoring system must obtain data from many distinct engine sensors, such as an oil temperature sensor, an oxygen sensor, etc. More sensors may be added in the future.

Recall instanceof

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {  
    long adj = 0;  
    if (acct instanceof CheckingAccount) {  
        checkingAcct = (CheckingAccount) acct;  
        adj = checkingAcct.getFee();  
    } else if (acct instanceof SavingsAccount) {  
        savingsAcct = (SavingsAccount) acct;  
        adj = savingsAcct.getInterest();  
    }  
    ...  
}
```

**Warning:
This code
is bad.**

- Advice: avoid instanceof if possible
 - Never(?) use instanceof in a superclass to check type against subclass

Recall instanceof

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {  
    long adj = 0;  
    if (acct instanceof CheckingAccount) {  
        checkingAcct = (CheckingAccount) acct;  
        adj = checkingAcct.getFee();  
    } else if (acct instanceof SavingsAccount) {  
        savingsAcct = (SavingsAccount) acct;  
        adj = savingsAcct.getInterest();  
    } else if (acct instanceof InterestCheckingAccount) {  
        icAccount = (InterestCheckingAccount) acct;  
        adj = icAccount.getInterest();  
        adj -= icAccount.getFee();  
    }  
    ...  
}
```

**Warning:
This code
is bad.**

Avoiding instanceof with the template method pattern

```
public interface Account {  
    ...  
    public long getMonthlyAdjustment();  
}  
  
public class CheckingAccount implements Account {  
    ...  
    public long getMonthlyAdjustment() {  
        return getFee();  
    }  
}  
  
public class SavingsAccount implements Account {  
    ...  
    public long getMonthlyAdjustment() {  
        return getInterest();  
    }  
}
```

Avoiding instanceof with the template method pattern

```
public void doSomething(Account acct) {  
    float adj = 0.0;  
    if (acct instanceof CheckingAccount) {  
        checkingAcct = (CheckingAccount) acct;  
        adj = checkingAcct.getFee();  
    } else if (acct instanceof SavingsAccount) {  
        savingsAcct = (SavingsAccount) acct;  
        adj = savingsAcct.getInterest();  
    }  
    ...  
}
```

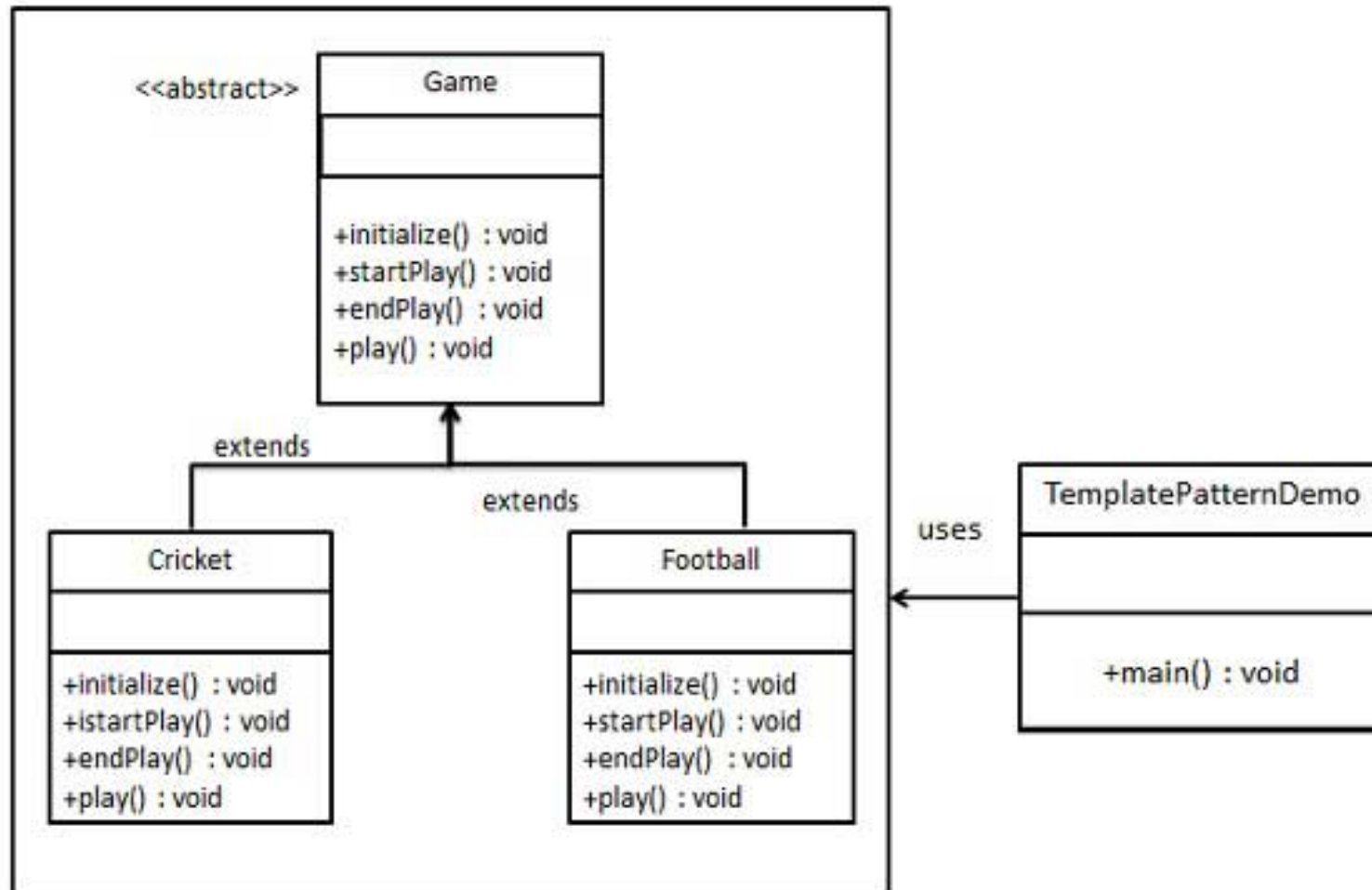
Instead:

```
public void doSomething(Account acct) {  
    long adj = acct.getMonthlyAdjustment();  
    ...  
}
```

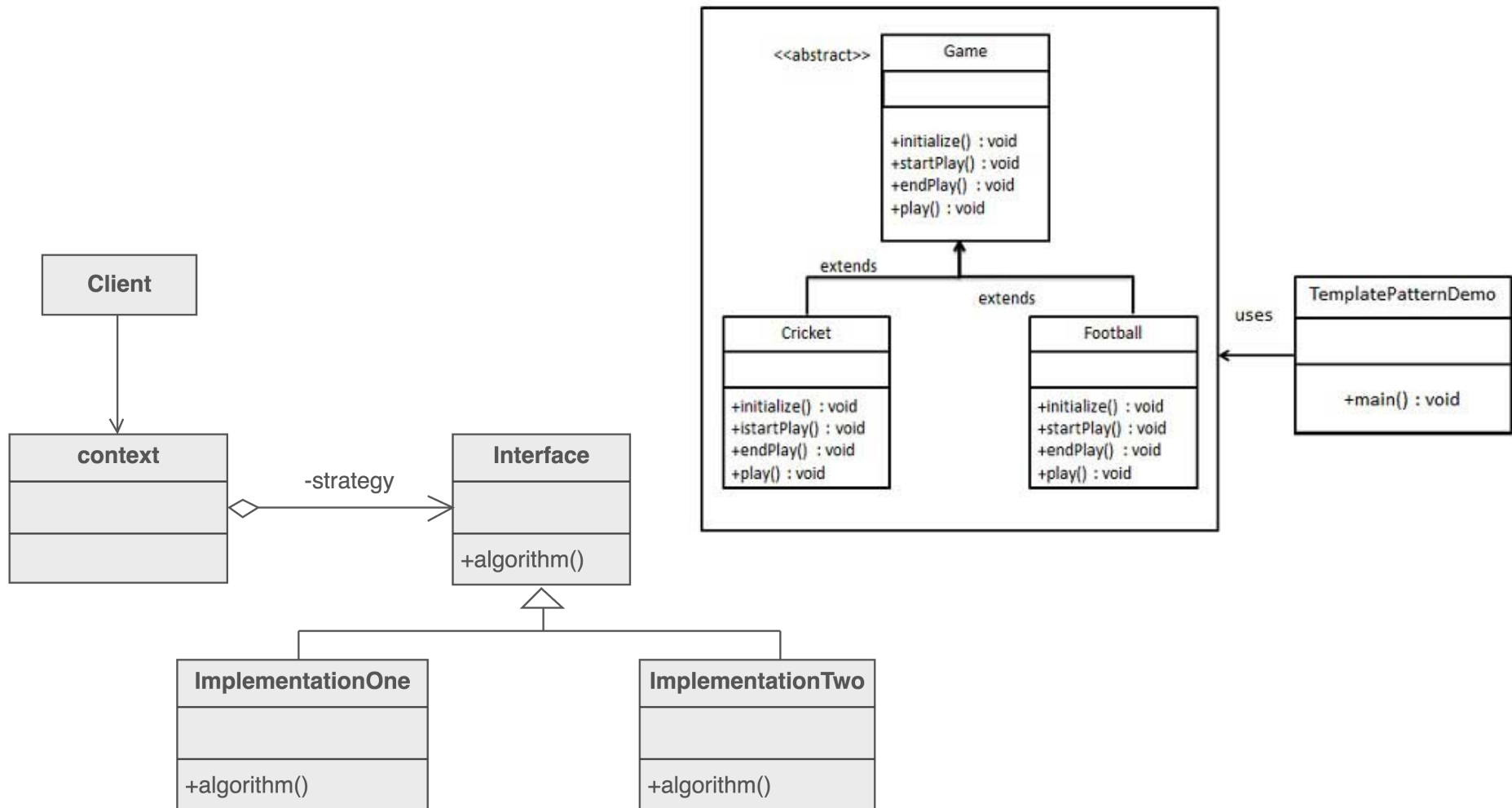
Template method pattern

- Problem: An algorithm consists of customizable parts and invariant parts
- Solution: Implement the invariant parts of the algorithm in an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations
- Consequences
 - Code reuse for the invariant parts of algorithm
 - Customization is restricted to the primitive operations
 - Inverted (Hollywood-style) control for customization

Template method UML



Strategy vs Template Method Patterns



Discuss: Strategy vs Template Method Pattern Usage

- What is an example where strategy would be a good fit?
- What is an example where Template Method would be a good fit?
- How are they different?