Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing Classes

UML + Design Patterns

Charlie Garrod **Michael Hilton**

School of Computer Science

institute for SOFTWARE RESEARCH

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

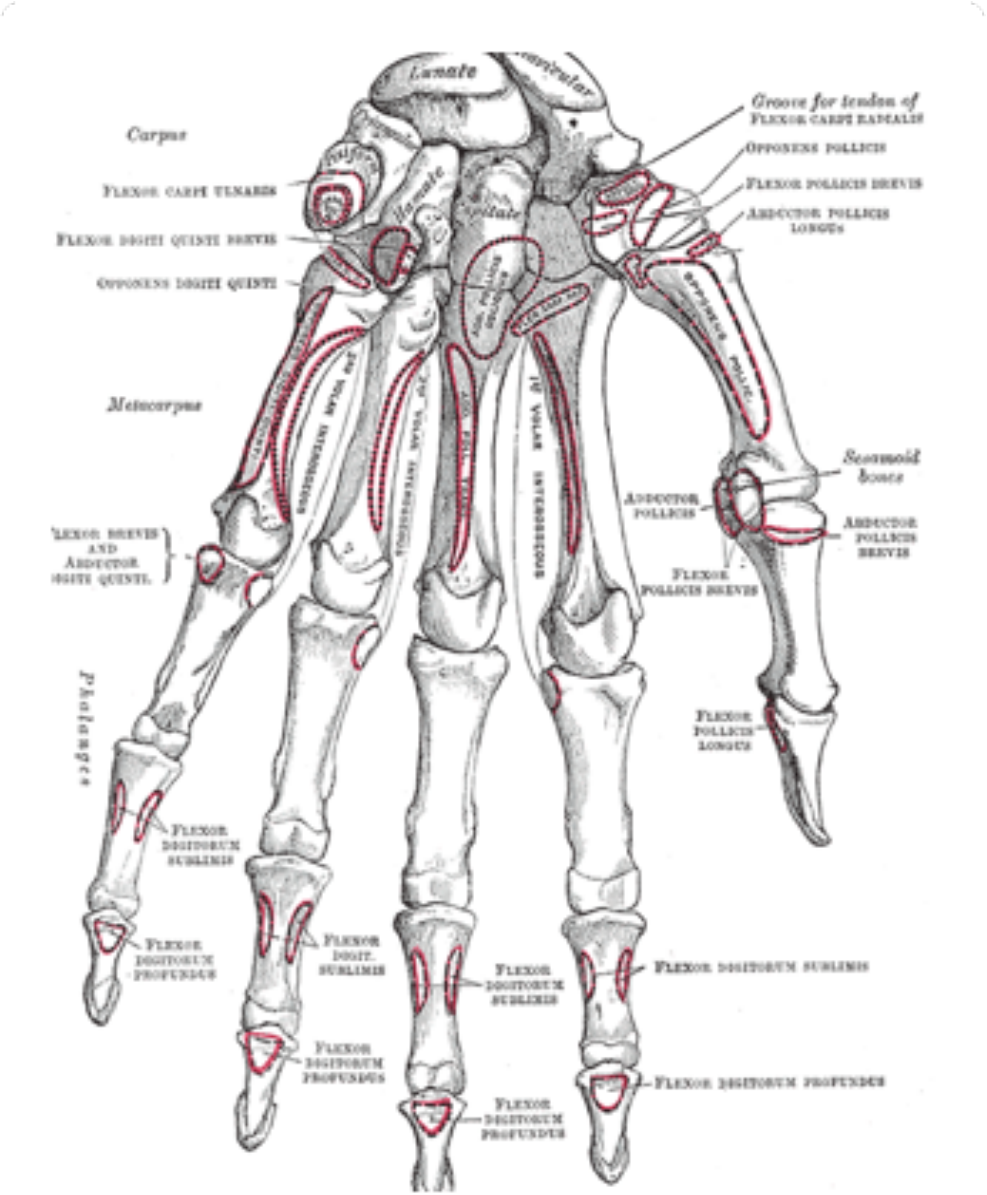
Are often useful when you need to:

Communicate,

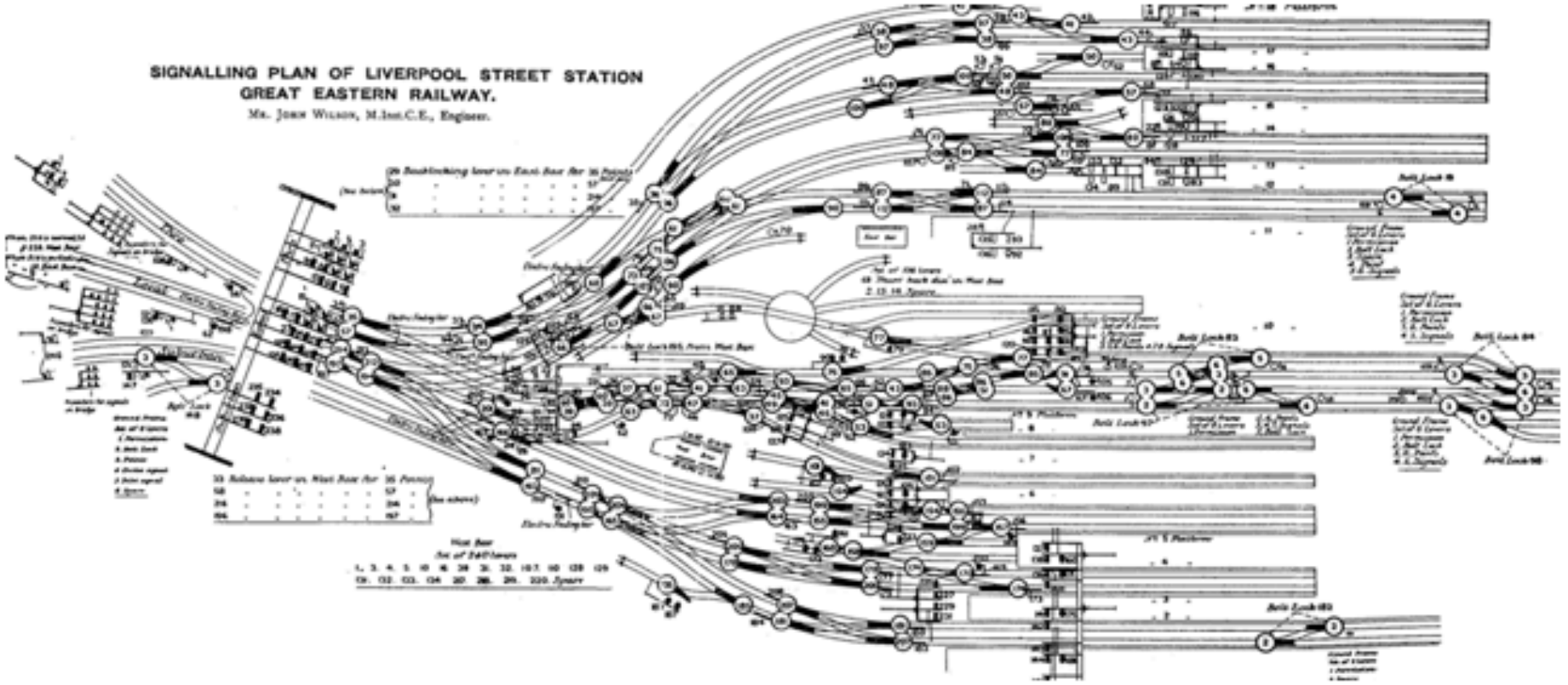
Visualize or

Analyze

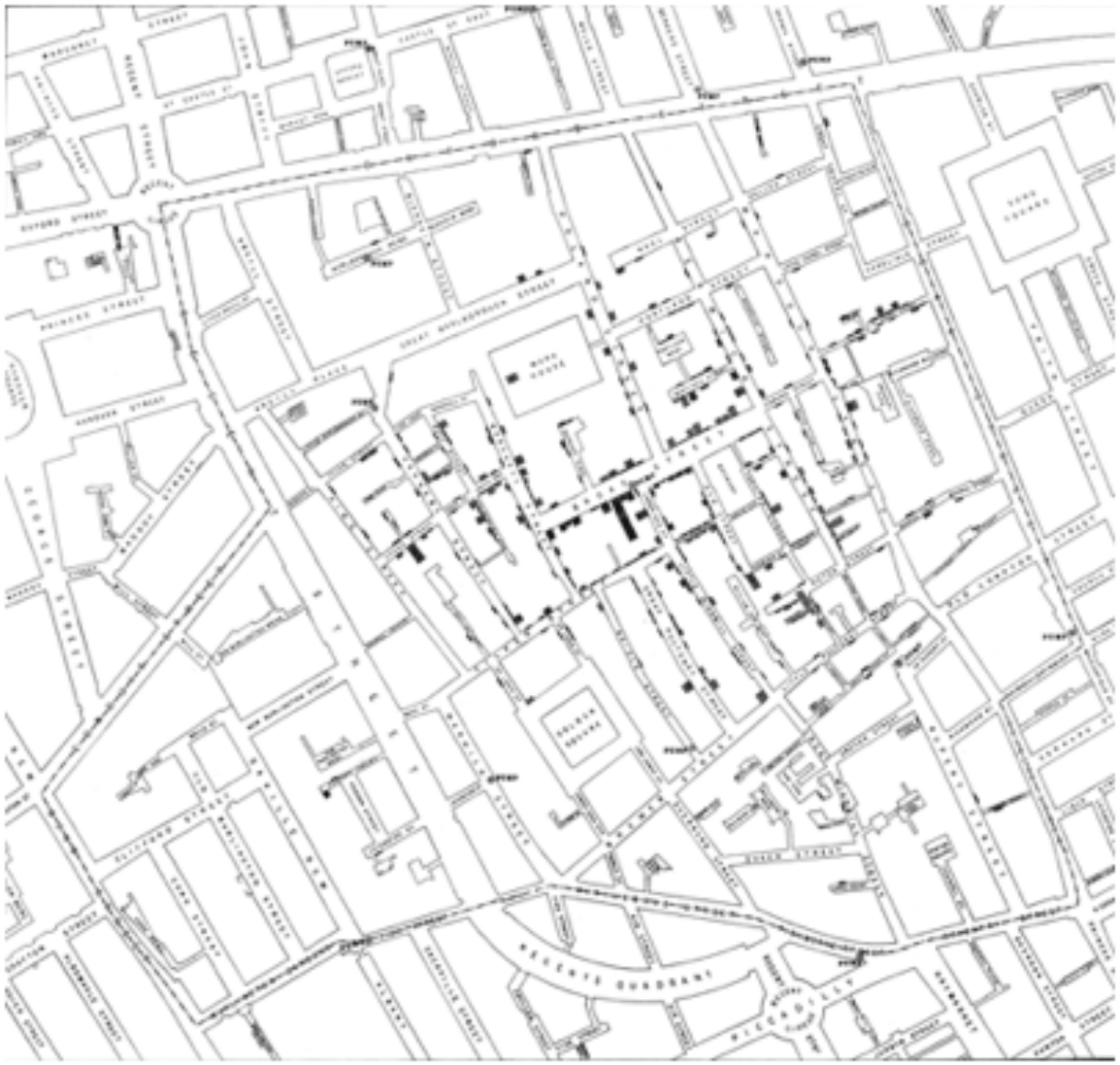
something, especially something with some structure.



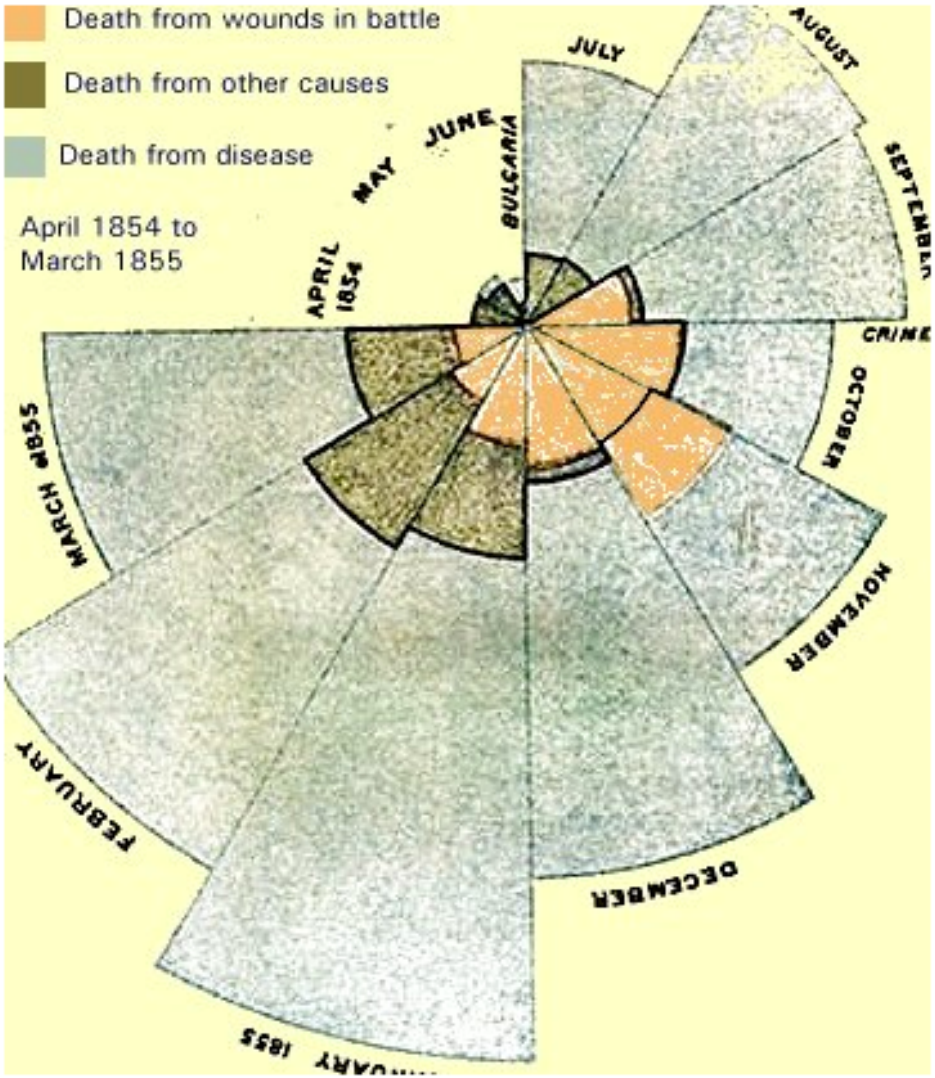
https://en.wikipedia.org/wiki/Gray%27s_Anatomy#/media/File:Gray219.png



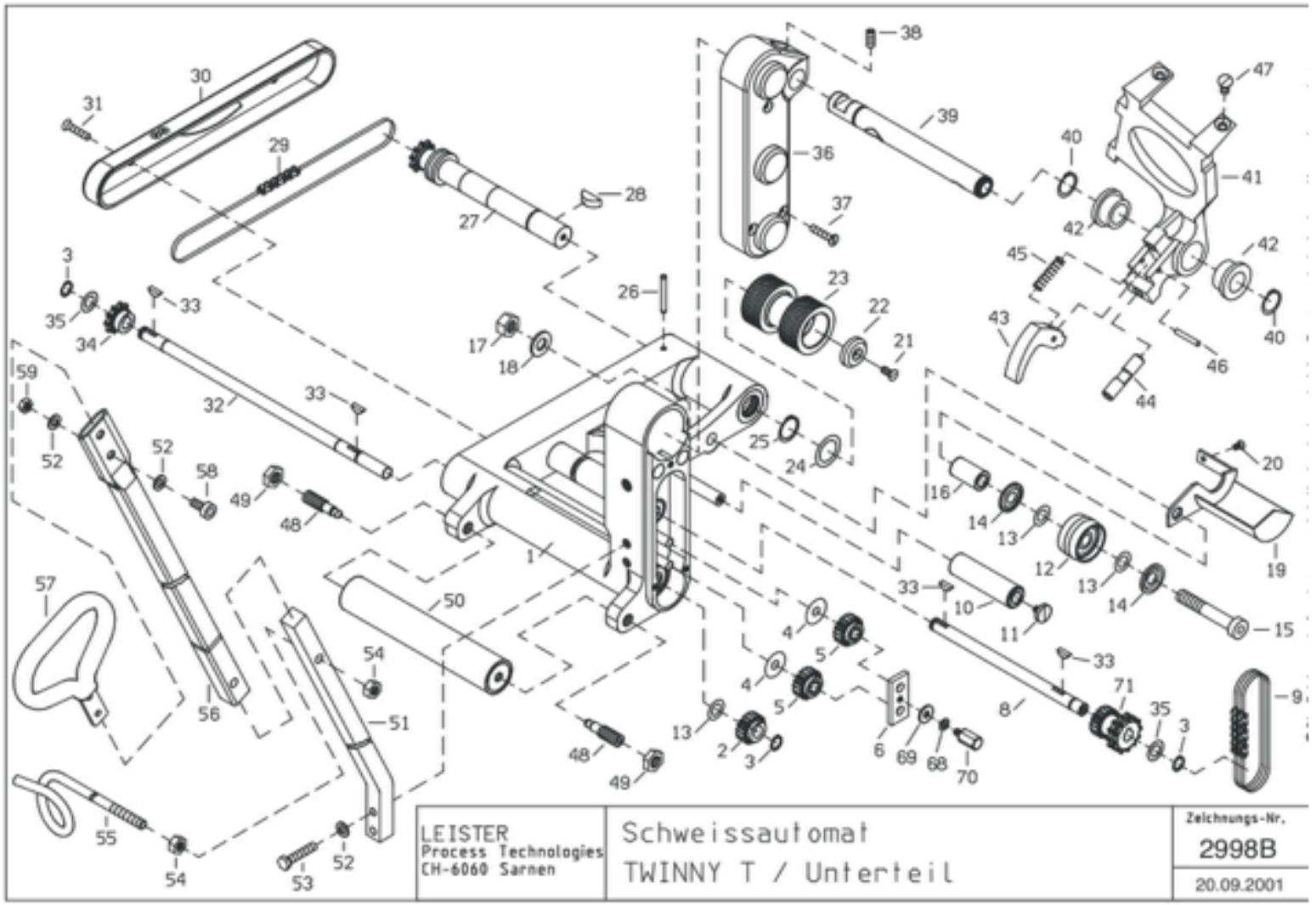
http://www.davros.org/rail/diagrams/old_liverpool_street.gif



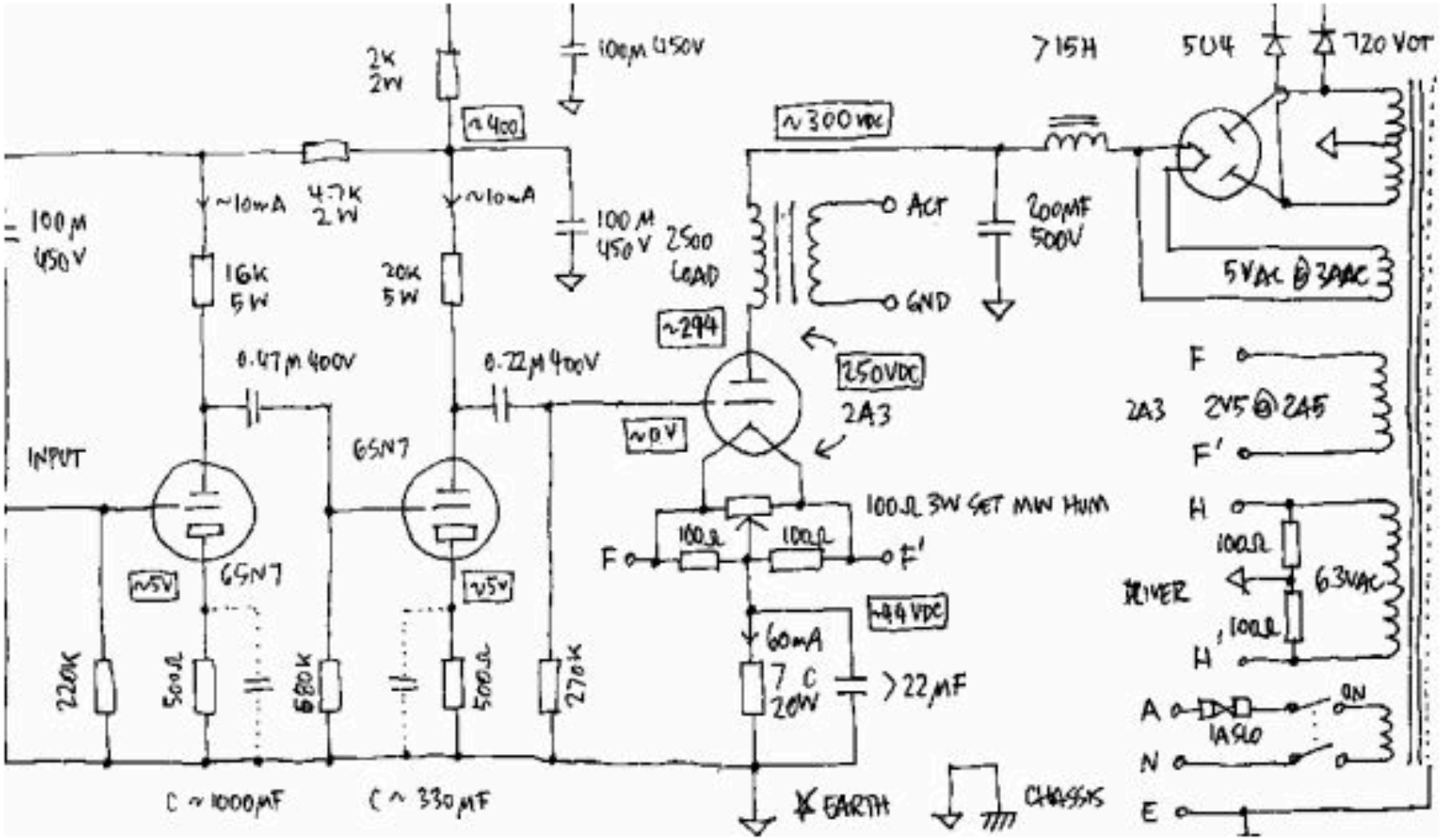
https://www.udel.edu/johnmack/frec682/cholera/snow_map.png



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



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

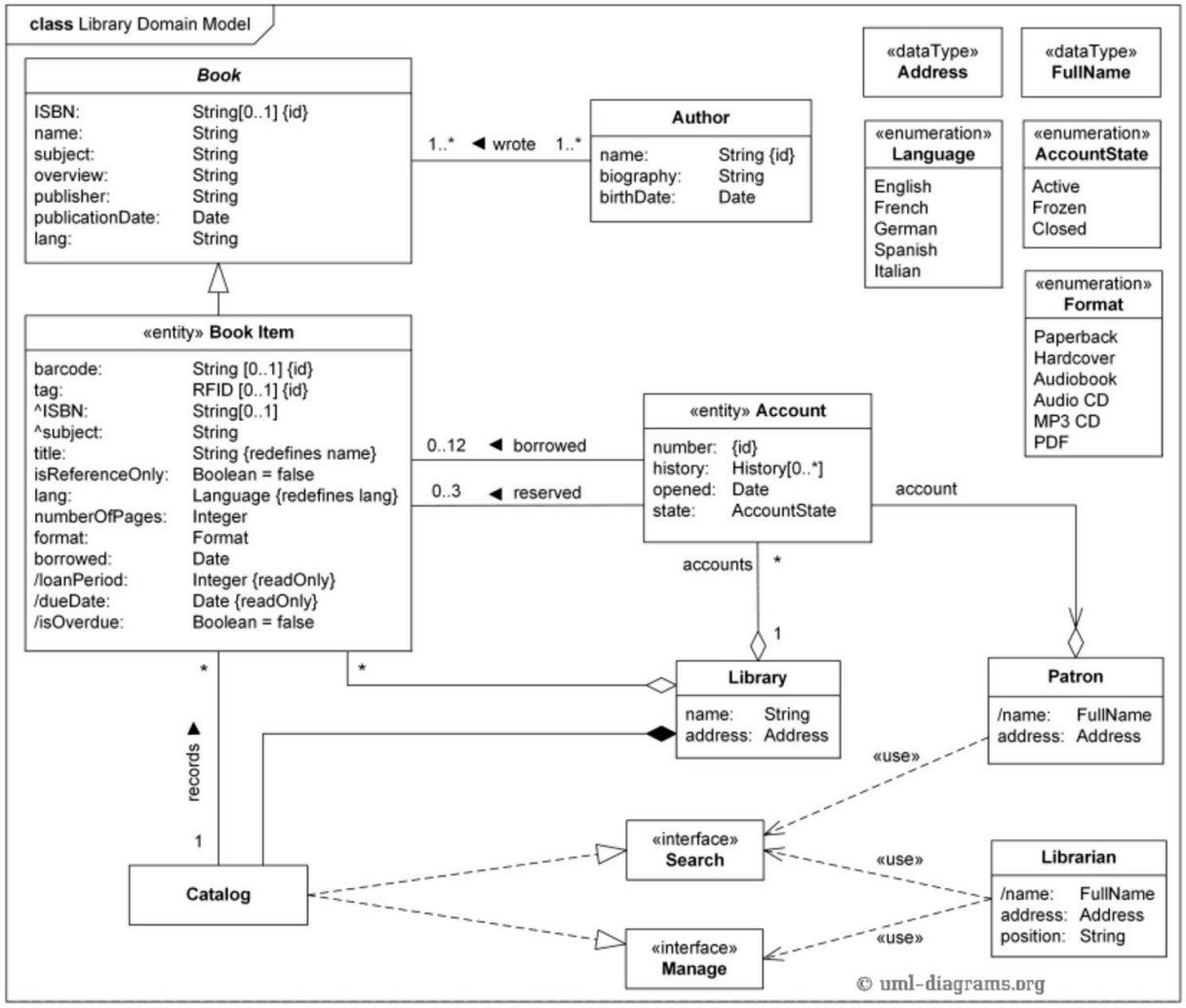


http://www.instructables.com/file/F7847TEFW4JU1UC/

"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)

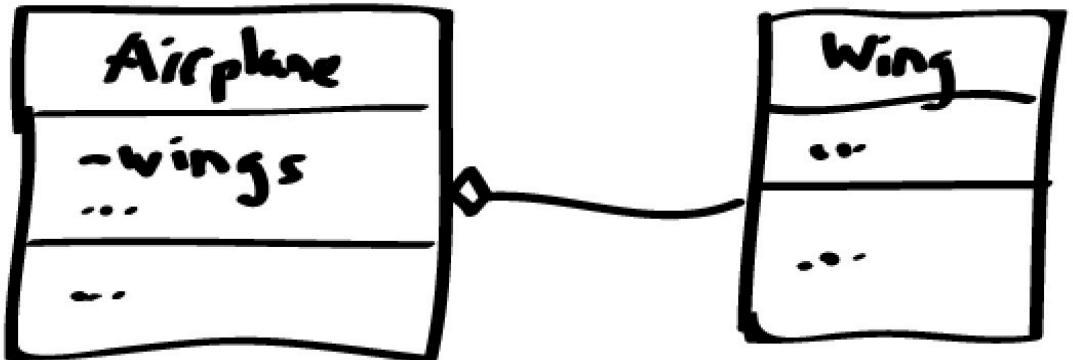
```
public abstract class AbstractAccount
        implements Account {
   protected long balance = 0;
   public long getBalance() {
        return balance;
   }
   abstract public void monthlyAdjustment();
   // other methods…
}

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

UML you should know

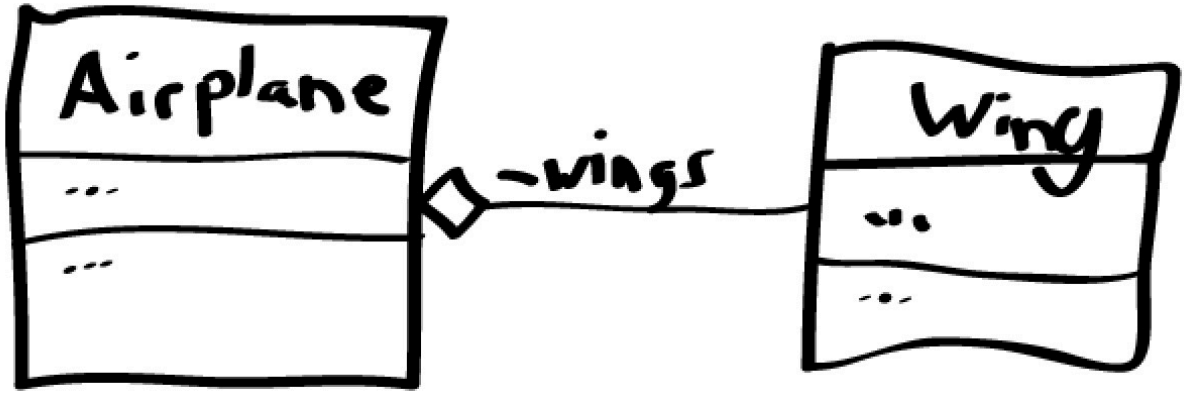
- Interfaces vs. classes
- Fields vs. methods
- Relationships:
  - "extends" (inheritance)
  - "implements" (realization)
  - "has a" (aggregation)
  - non-specific association
- Visibility: + (public) - (private) # (protected)
- Basic best practices…

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 Processer {
    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

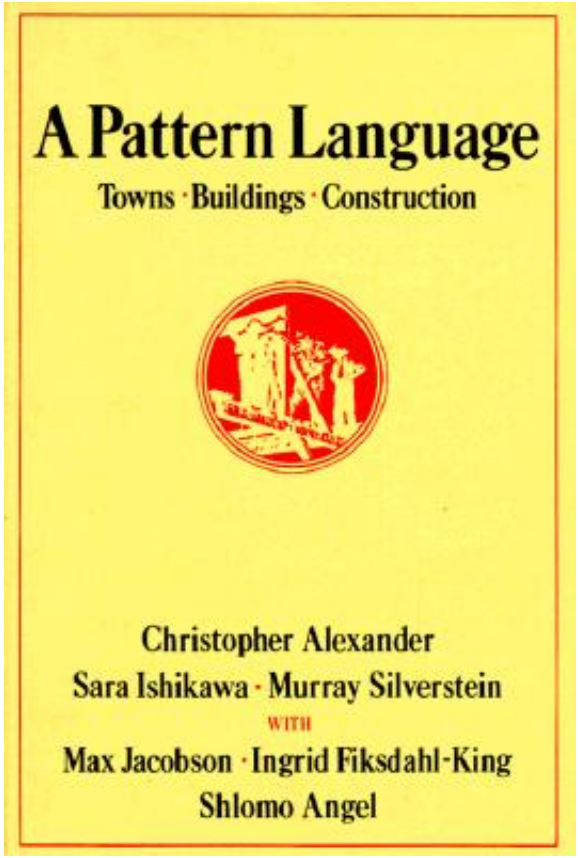
- Suppose we need to sort a list in different orders…

```
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)



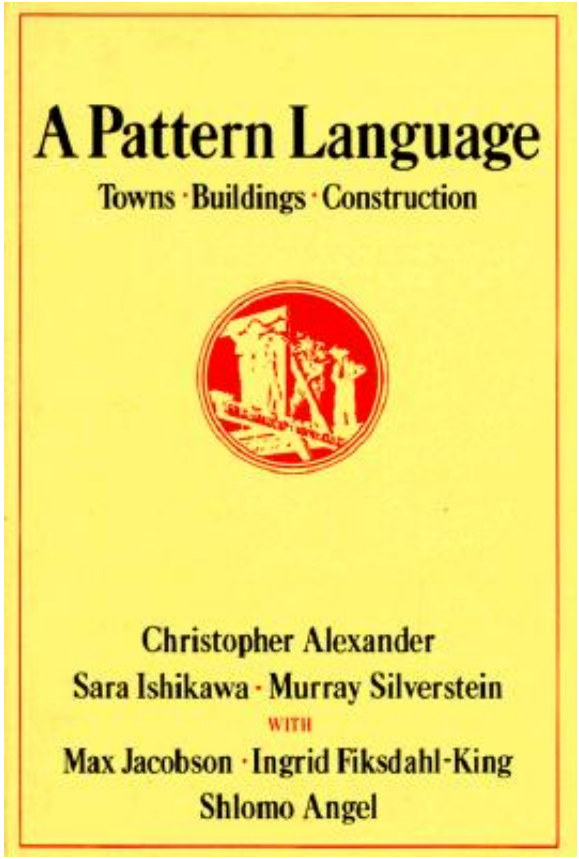
DESIGN PATTERNS

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.



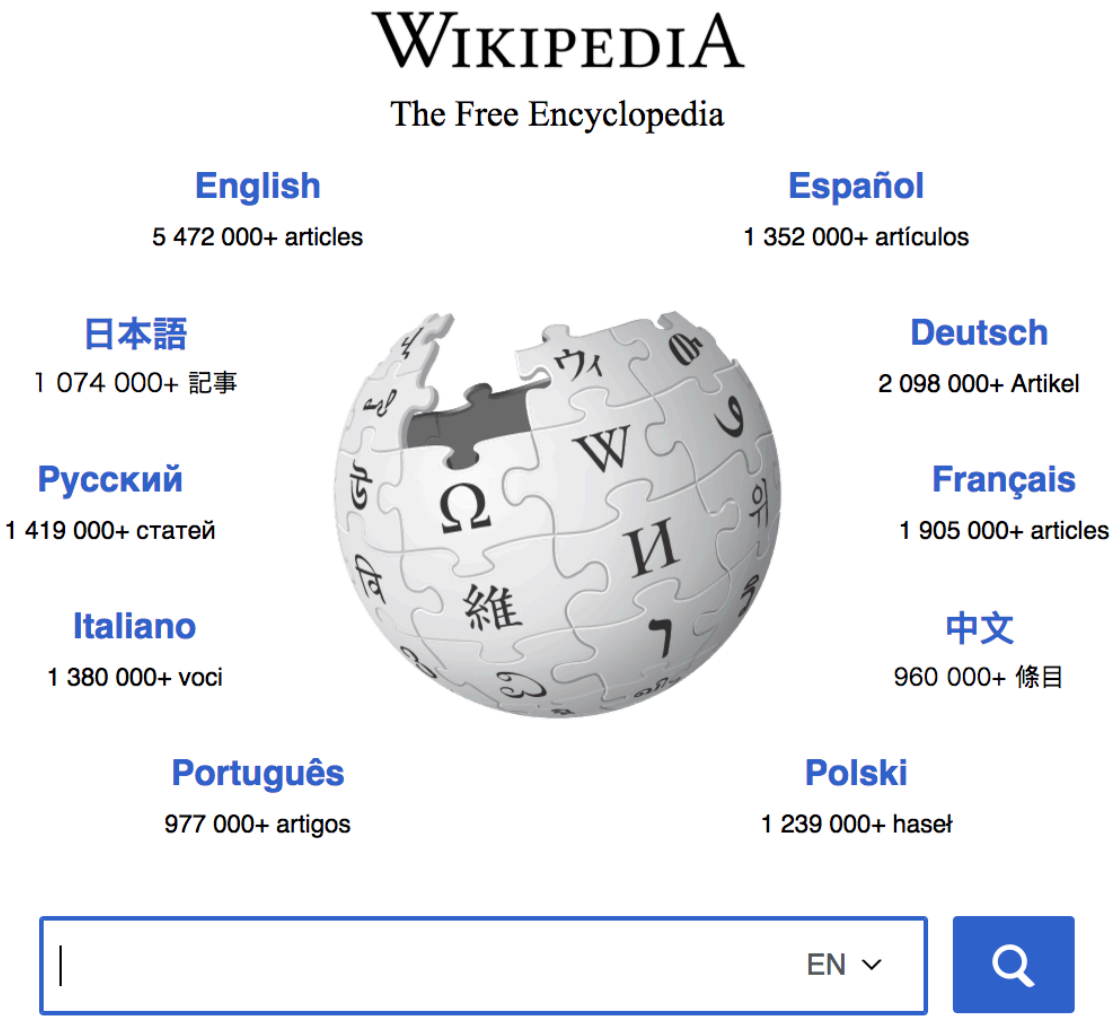
The timeless Way of Building

- 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

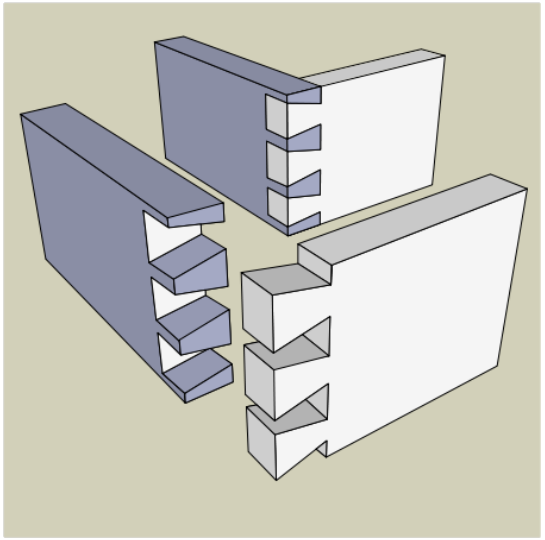
- Are there problems in software that occur all the time that can be solved in somewhat the same manner?
- Is it possible to design software in terms of patterns?

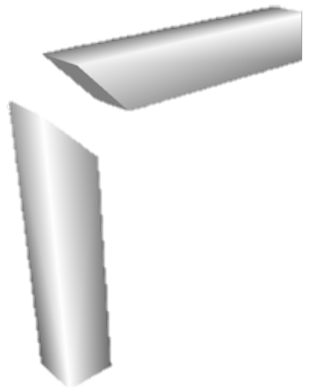
How not to discuss design (from Shalloway and Trott)

- 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…

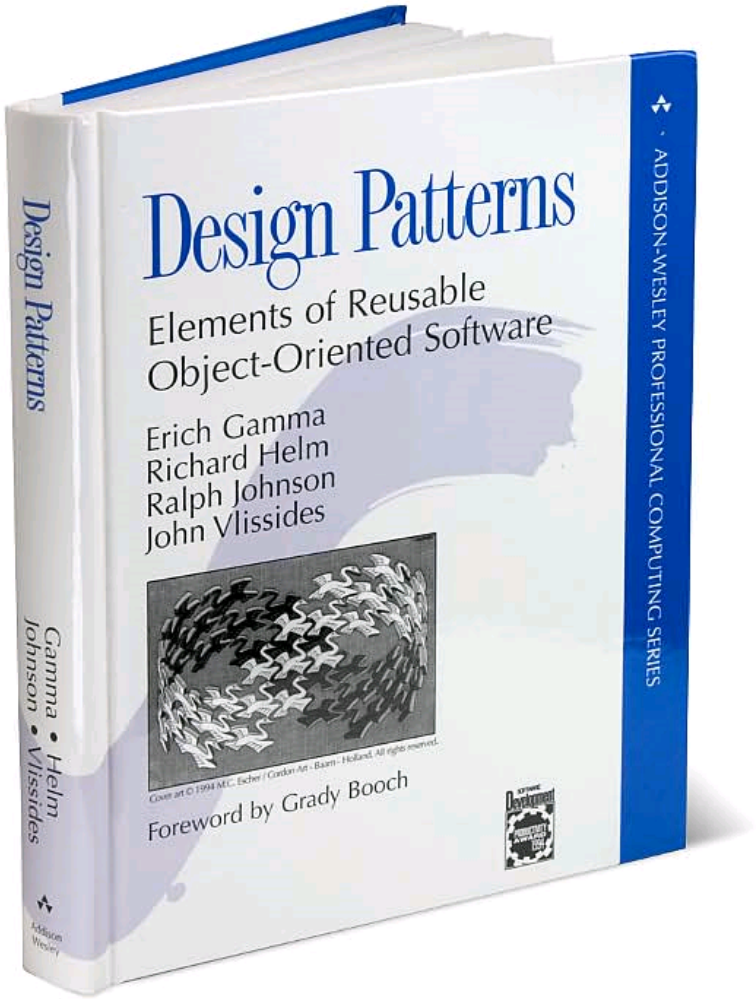
Discussion with design patterns

- 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

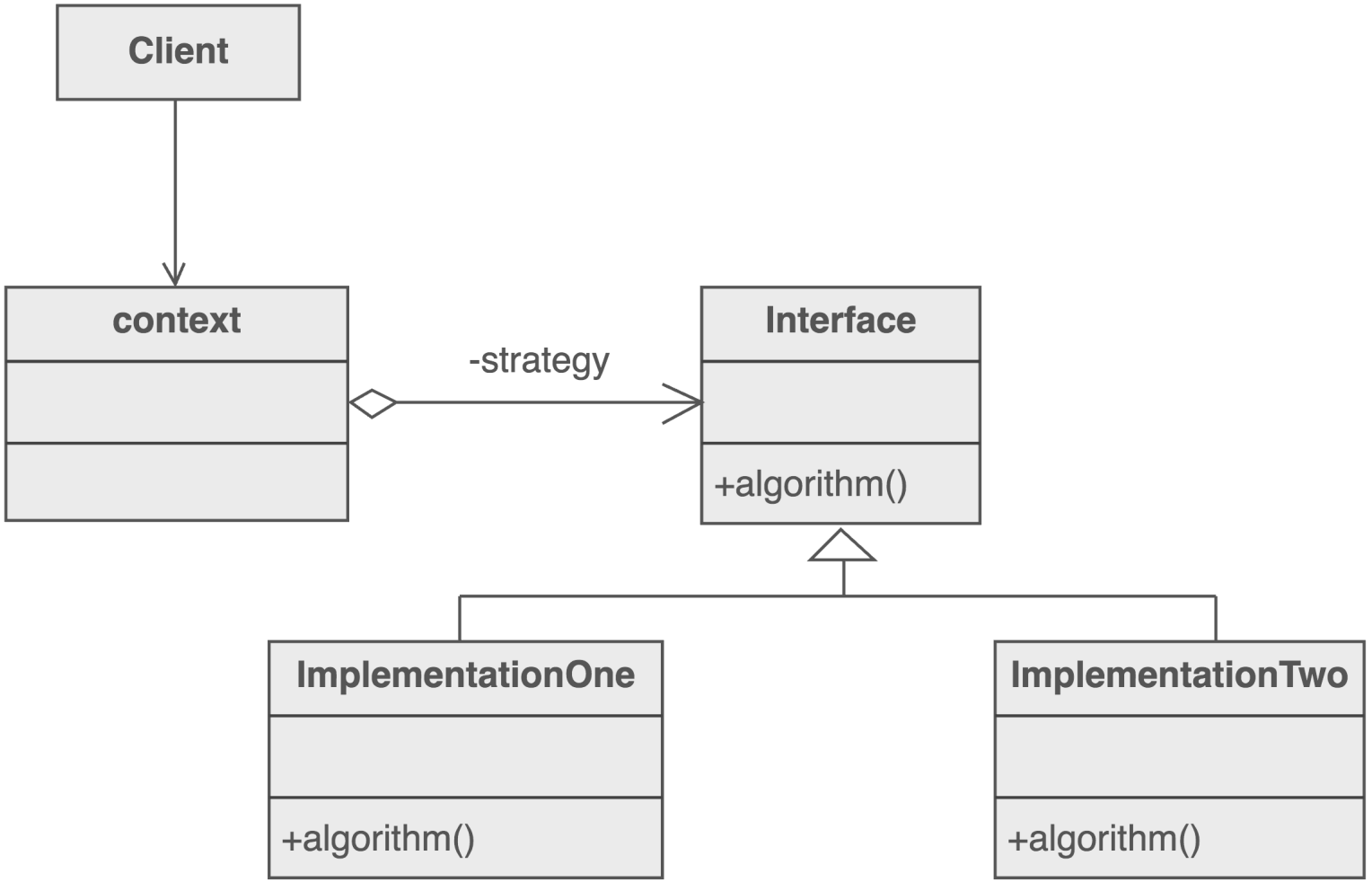
Recognizing a pattern

- 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();
  }
  …
}
```

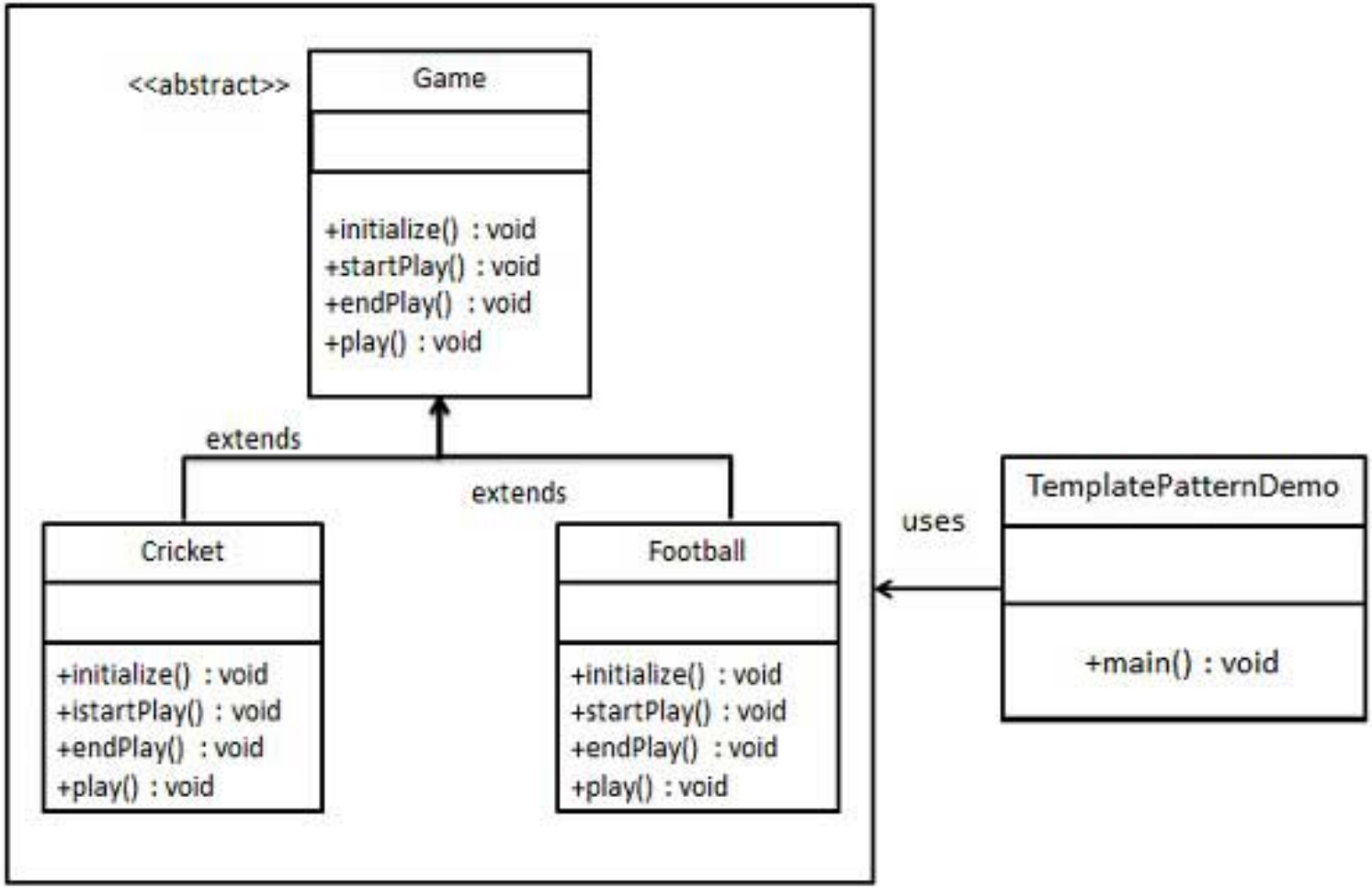
Instead:

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

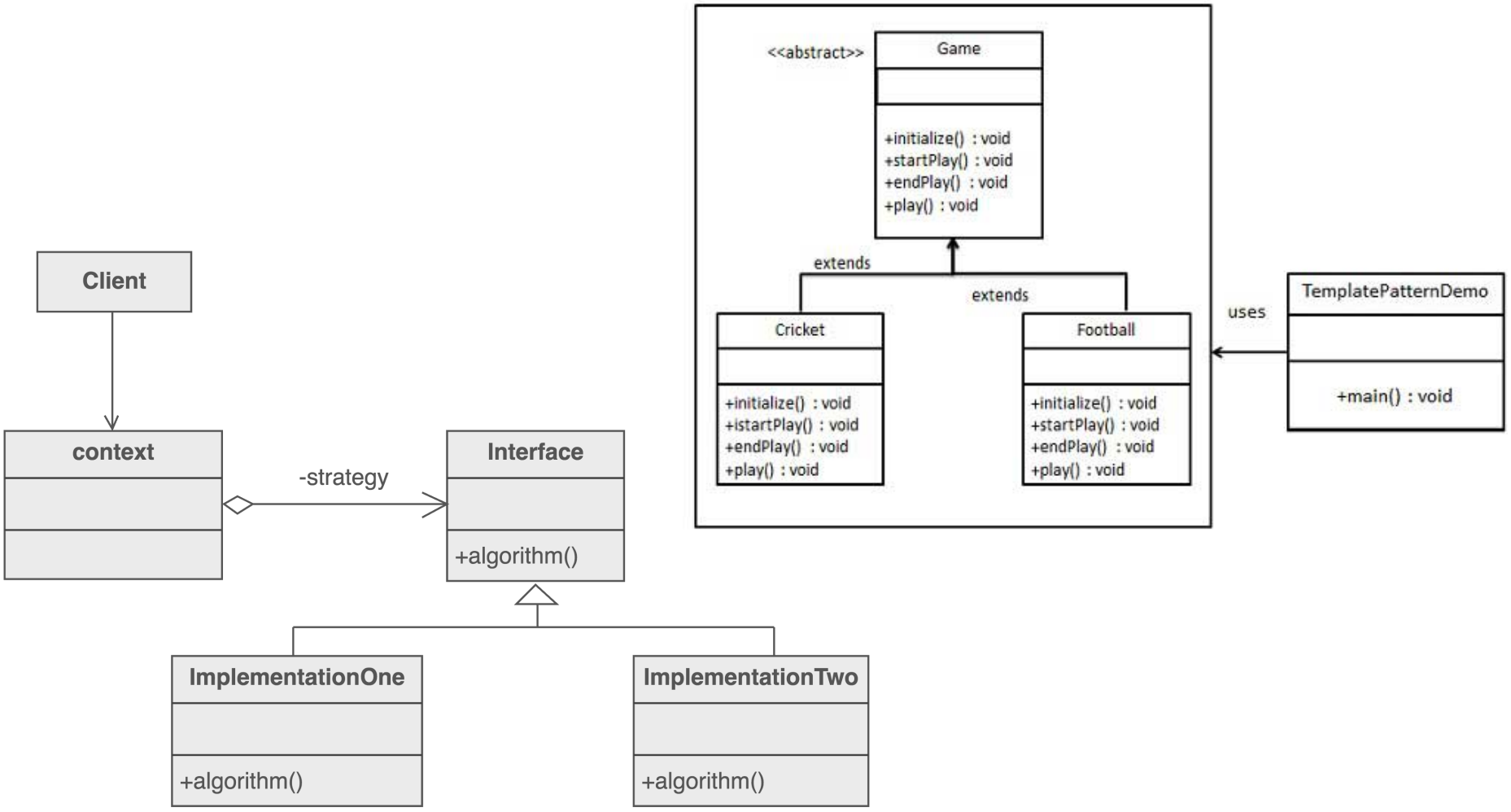
Template method pattern

- 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?