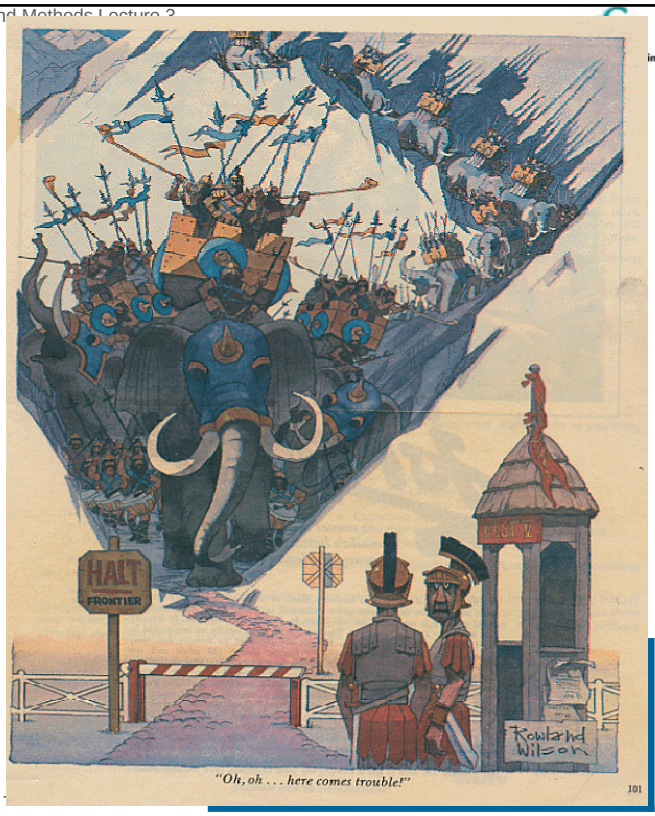
Software engineering revisited



Exercise 5 - testing "SumPrimes"

CERN School of Computing

Should "len" be included or not?

Its OK for a prime number to be divisible by one

```
int sumPrimes(int len) {
  int sum = 0;
  for (int i=1; i<len; i++) { //loop over possible primes
    bool prime = true;
    for (int j=1; j<10; j++) { //loop over possible factors
      if (i % j == 0) prime = false;
    }
    if (prime) sum += i;
  }
  return sum;
}
```

If you divide a number by itself, the remainder is zero

Lesson 1: Its not easy to understand somebody else's code

- **Assumptions, reasons are hard to see**
  - **"Is one a prime number?"**
  - **Test defines the behavior!! assertTrue(sumPrimes(1)==1)**

Lesson 2: Better structure would have helped

- **Separate "isPrime" from counting loop to allow separate understanding**
- **Make the algorithm for checking prime even clearer**

Exercise 5 - isCube, isSquare, et al

New bugs:

- **Just introduced**
- **Newly discovered in another area**
- **Newly understood to be bugs**

Too many possibilities, how do you keep track?

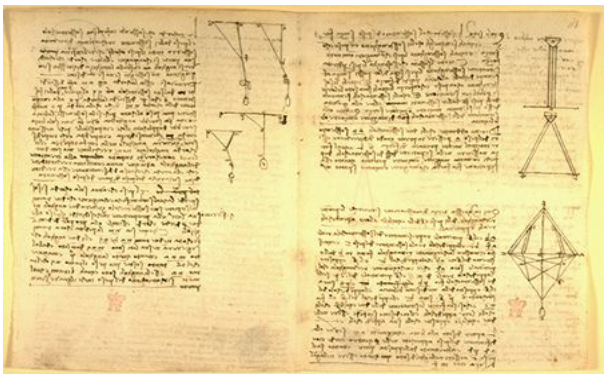
This is why large projects get harder as you go along!

Design

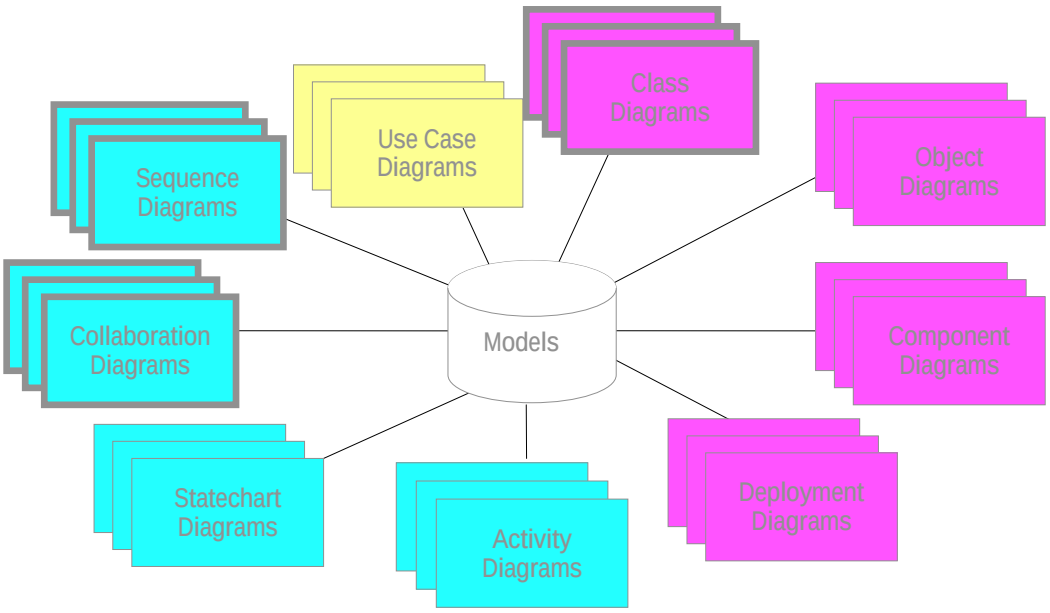
Specify the details of inter-object collaboration *mechanisms*

- **Determine the *structure* of classes and their associations**
  - **Relationships of access, ownership, authority**
- **Determine the *behavior* of classes**
  - **E.g. Interactions with other objects**
    - **Collaboration**
    - **Sequence**

How do we record and communicate this?

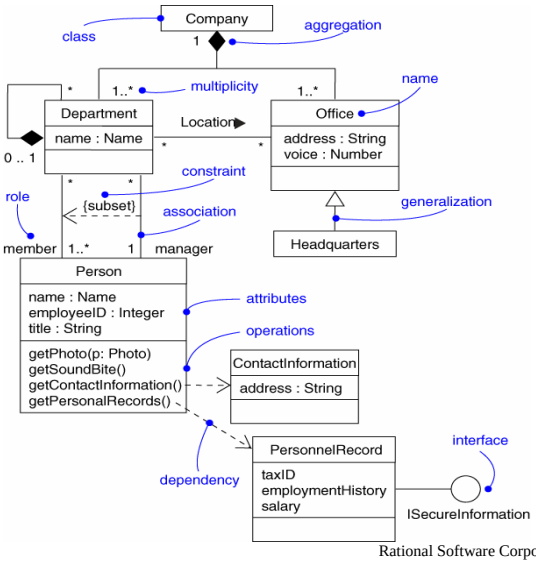


UML Diagrams



Class Diagram

Describes the types of objects in the system and the various kinds of static relationships that exist between them

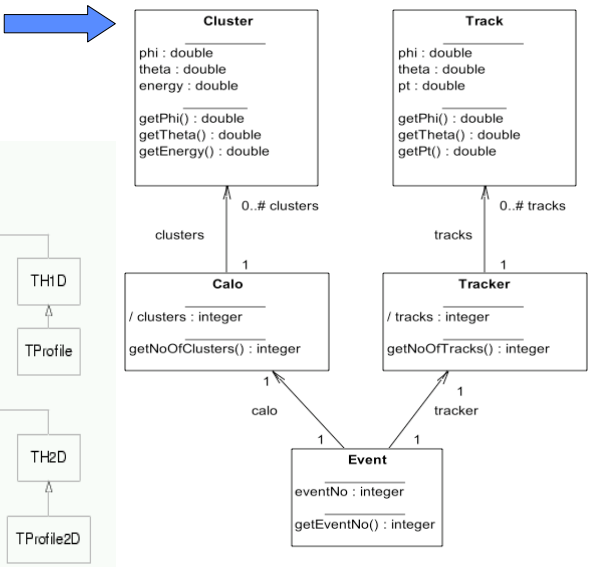


Rational Software Corporation

Example Class Diagrams

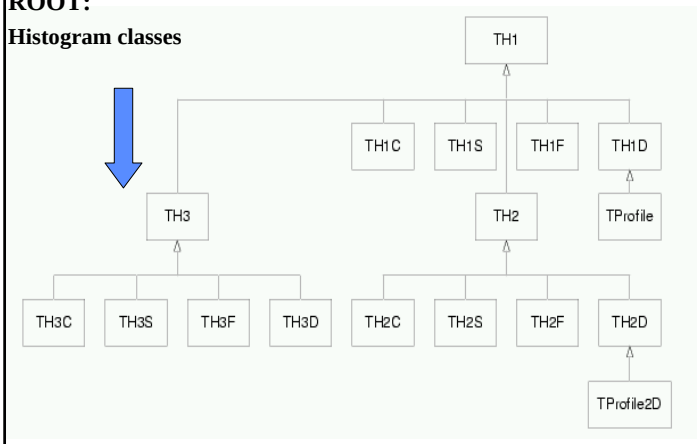
LHC++/Anaphe:

Event structure as defined in DDL file for populateDb exercise



ROOT:

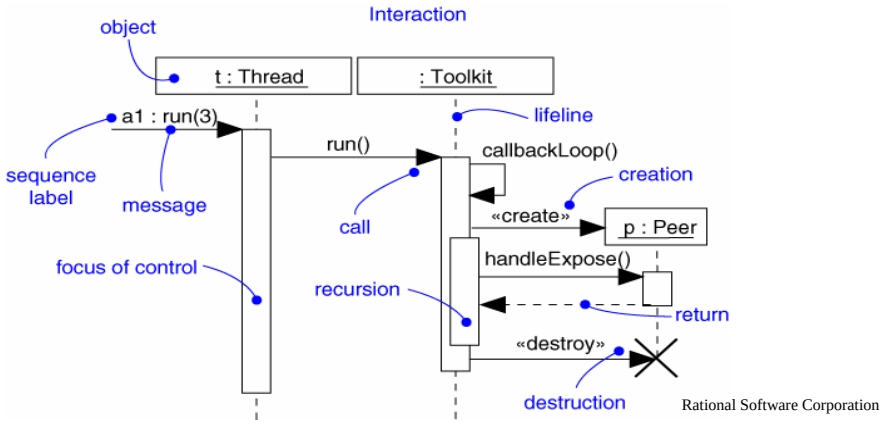
Histogram classes



Sequence Diagram

Captures dynamic behavior (time-oriented)

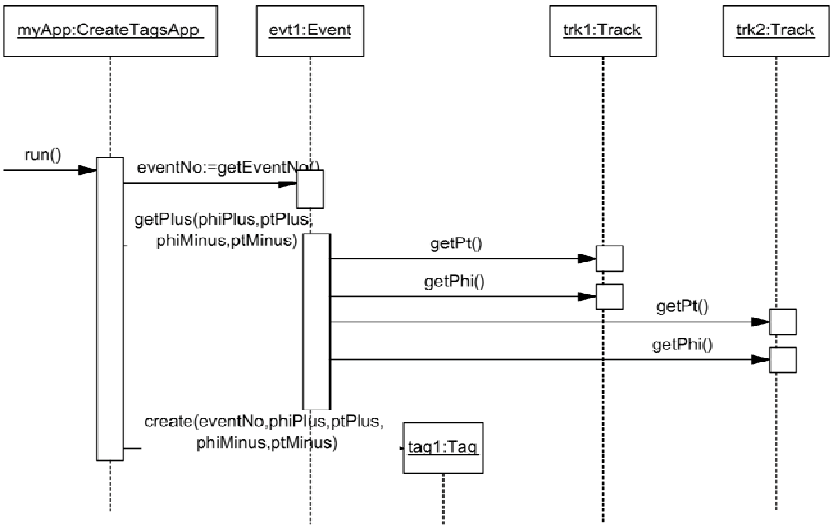
- **Model flow of control**
- **Illustrate typical scenarios**



Rational Software Corporation

Example Sequence diagram

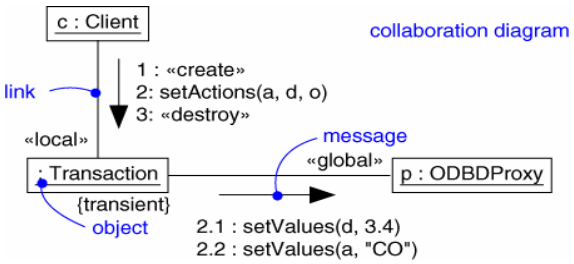
LHC++/Anaphe: scenario for createTag exercise with 1 event and 2 tracks



Collaboration Diagram

Captures dynamic behavior (message-oriented)

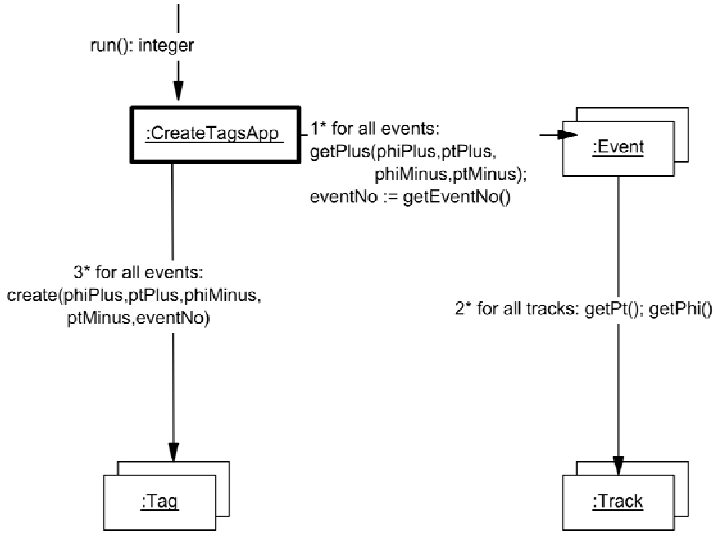
- **Model flow of control**
- **Illustrate coordination of object structure and control**



Rational Software Corporation

Example Collaboration Diagram

LHC++/Anaphe: messages between classes for CreateTag exercise



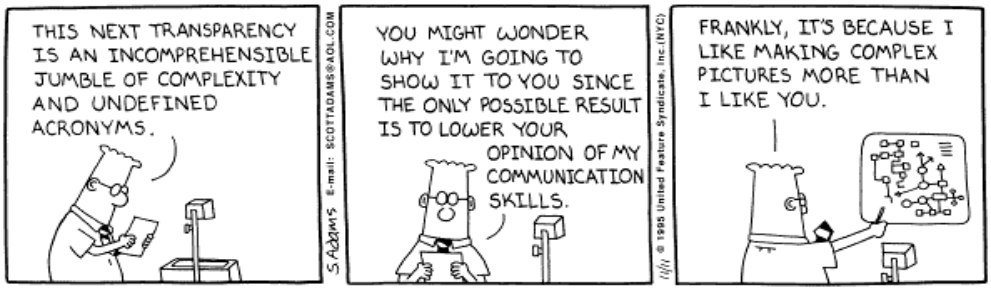
"These are complicated"

"So is field theory"

•Which is physicist-speak for "I don't get it either, so I'll call it 'trivial'"

"It's just notation"

•The notation is complicated because it's representing a complicated thing



"Yes, and how do we know they're right?"

- **That's the key question.**

Example: Linear Algebra

Physics code contains lots of linear algebra: A*X+B

- **Where A, X and B are more than just numbers: vectors, matrices**

Complicated operations:

- **Only some operations are OK**
  - **Can't add, dot-product vectors of different sizes**
  - **Dimensions must agree for vector-matrix multiplication**
- **But within those rules, users don't want to care about restrictions**
  - **A measurement might be a 1D, 2D or 3D constraint, but same formula to use it**

What are the trade-offs for a "linear algebra library"? For users?

- **Time & space of the linear algebra code**
- **Ease of use**
- **Time & space of the using code**
- **Correctness of answers**

Implementation: Vector3, Matrix32

```
class Vector3 {
float values[3];
float dotWith(Vector3 v) {...}
Vector3 add(Vector3 v) {...}
...
}
```

```
class Matrix33 {
float values[3,3];
Vector3 multiply(Vector3 v) {...}
Matrix33 add(Matrix33 v) {...}
...
}
```

Does the job

- Once you've created one of these, you can just string together operations
- Code to implement each method is quite simple

Does the job

- **Once you've created one of these, you can just string together operations**
- **Code to implement each method is quite simple**

But needs lots and lots and lots of methods

- **Vector3 can multiply Matrix32, Matrix33, Matrix34, Matrix35, Matrix36, ...**
- **Similar numbers for matrix multiplication**
- **Large amount of duplicated code to make a general library**

Can we get smarter with inheritance?

- **Matrix class, with Matrix32, Matrix33, Matrix34 as subclasses**
- **Methods then take and return Matrix objects**

Problem: Implementation of methods still needs to know

- **Methods require size information, access to individual elements**
  Different size internal arrays need to be accessed, compiler wants to know
- **Lots of work to get those**
- **And methods still need to call "new Matrix32" vs "new Matrix33", etc**

General Implementation: Vector, Matrix

```
class Vector {
int dim;
float *values[dim];
float dotWith(Vector v) {...}
Vector add(Vector v) {...}
...
}
```

```
class Matrix {
int dim1, dim2;
float *values[dim1, dim2];
Vector multiply(Vector v) {...}
Matrix add(Matrix v) {...}
...
}
```

Again does the job

- Once you've created one of these, you can just string together operations
- Code to implement each method is almost as simple
- "Just has to" keep track of index dimensions, and do one indirection
- Return types are fixed, so only need to handle one "new"

Strong, general approach for a library, but at what cost?

Costs:

Tradeoffs:

Direct structure

Minimal memory use:

Compiler handles limits, allocates data as part of object

Fast allocate/deallocate:

Vector[5] is just one long allocation & 5 ctor calls

More complicated user code:

You have to explicitly specify classes for intermediate variables, etc; can't pass common super-types

Indirect structure

More memory needed:

Virtual table pointer

Length values

Pointer to memory

Allocate/deallocate is more work:

Vector[5] is one allocation, 5 ctor calls, then 5 more allocations

User code simple, general:

All objects are same basic type

Code can be written without reference to specific sizes

When there's no perfect answer, you're in the realm of tradeoffs

Start with the general, and replace with specific when needed?

This is where iterative development comes in…

Imagine the project is not to build software but a bridge…

Initial Requirements: A to B

Successful Development Program!

Analogy shows successful iterations:

- **The basic *product* existed from the first iteration and met the primary requirement: A to B**
- **Early emphasis on defining the architecture**
- **Basic architecture remained the same over iterations**
- **Extra functionality/reliability/robustness was added at each iteration**
- **Each iteration required more analysis, design, implementation and testing**
- **Use case (requirements) driven**
  - does what the users want - not what the developers think is cool

Some limits to analogy:

It took people centuries to figure out how to build big bridges

And we developed engineering processes to do the big ones!

Little of the early cycles survived in final one

How to pick what goes in the next iteration?

Choice of additions for an iteration is *risk driven*

- **Early development focuses on components with the highest risk and uncertainty**
  - **Avoids investing resources in a project that is not feasible**
- **But it has to do something basically useful**
  - **So all involved will take it seriously**

What can go wrong?

Advantages of Iterative and Incremental Development

Complexity is never overwhelming
- **Only tackle small bits at a time**
- **Avoid *analysis paralysis* and *design decline***

Early feedback from users
- **Provides input to the analysis of subsequent iterations**

Developers skills can *grow* with the project
- **Don't need to apply latest techniques/technology at the start**
- **Get used to delivering finished software**

Requirements can be modified
- **Each iteration is a mini-project (analysis, design….)**

Note that these benefits come from completing, deploying and using the iterations!

Lecture summary

Software engineering is the art of building complex computer systems

It's ideas and techniques spring from our need to handle size & complexity

As you do your own work & develop your own skills, consider:
- **How your effort effects or contributes to things 10X, 100X, 1000X larger**
- **How you'll do things different/better when it's your problem**