# Week 2
# Modules: 1 and 2

## Starting Berkeley Time

**Topics:**

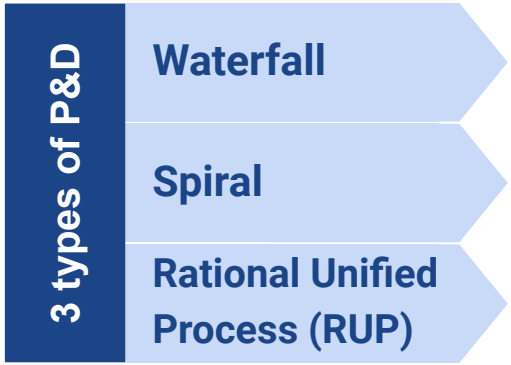
- Software as a Service, Agile Development, and Cloud Computing
- Ruby Review

# Software as a Service, Agile Development, and Cloud Computing

# Plan & Document Software Development Process

**General Idea**

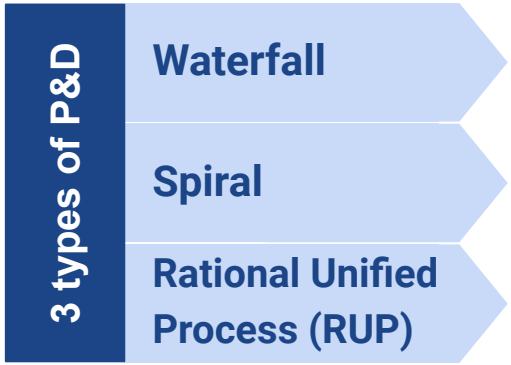
- Before development, come up with a project plan, including an extensive, detailed documentation to improve predictability
- Progress is measured against the plan
- Documentation is necessary so that
  - New people can be onboarded
  - Information is not lost



# Plan & Document Software Development Process

**General Idea**

- Before development, come up with a project plan, including an extensive, detailed documentation to improve predictability
- Progress is measured against the plan
- Documentation is necessary so that
  - New people can be onboarded
  - Information is not lost



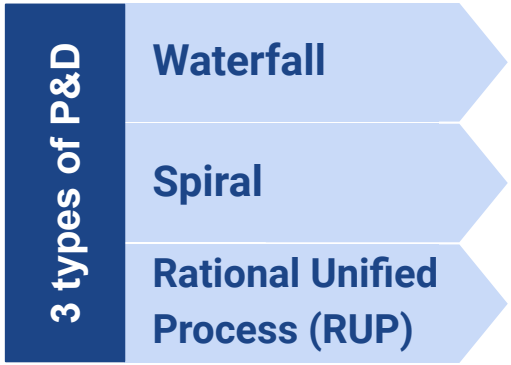

**How did it come into existence?**

- Problem: Unplanned Software Development => Unpredictability

# Plan & Document Software Development Process

**General Idea**

- Before development, come up with a project plan, including an extensive, detailed documentation to improve predictability
- Progress is measured against the plan
- Documentation is necessary so that
  - New people can be onboarded
  - Information is not lost



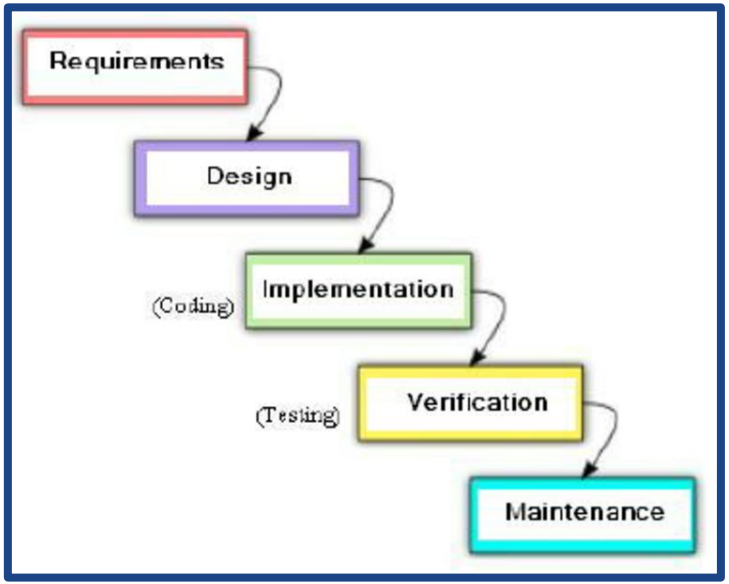

**How did it come into existence?**

- Problem: Unplanned Software Development => Unpredictability
- Need: build software that was predictable in quality, cost & time
  - Inspiration from bridge construction in civil engineering

# Plan & Document: Waterfall

**General Idea: A top-down approach**

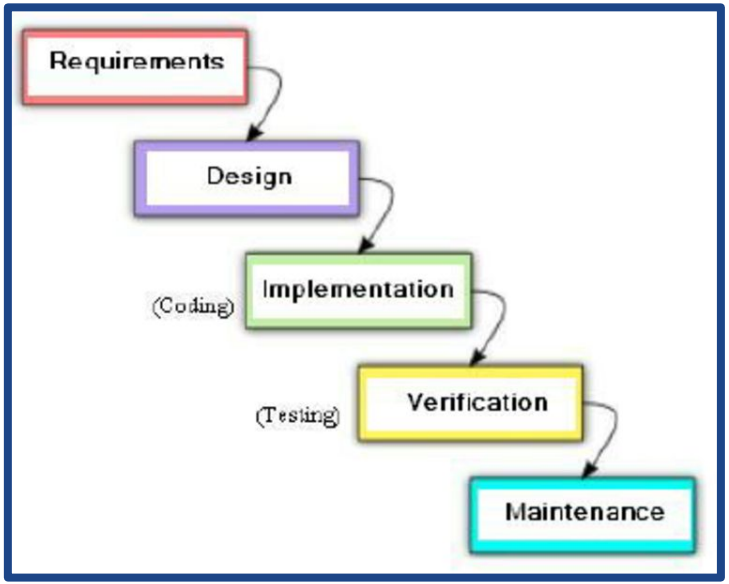
- Philosophy: complete a phase before going on to the next one (one-directional)
- Rational
  - earlier you find an error the cheaper it is to fix
  - removing as many errors as early as possible
  - This prevents unnecessary work



# Plan & Document: Waterfall

**General Idea: A top-down approach**

- Philosophy: complete a phase before going on to the next one (one-directional)
- Rational
  - earlier you find an error the cheaper it is to fix
  - removing as many errors as early as possible
  - This prevents unnecessary work



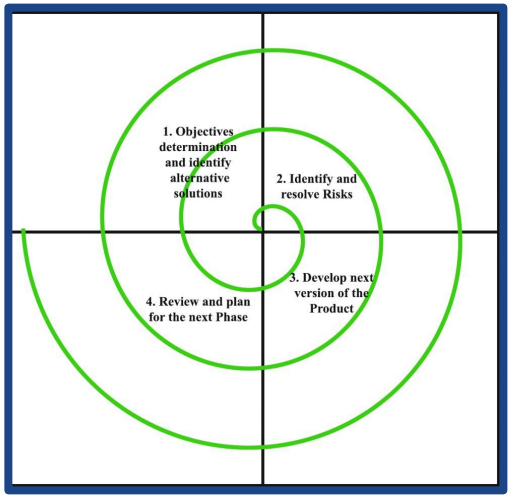

**Problem**

- This strategy  runs into a trouble when customers change their minds about what they want, especially if they do so later during the process

# Plan & Document: Spiral

**General Idea: Prototyping + Waterfall**

- The idea is to iterate through a sequence of four phases, with each iteration resulting in a prototype that is a refinement of the previous version.
- 4 phases
  - Determine objectives and constraints of this iteration
  - Evaluate alternatives; identify and resolve risks
  - Develop and verify the prototype for this iteration
  - Plan the next iteration



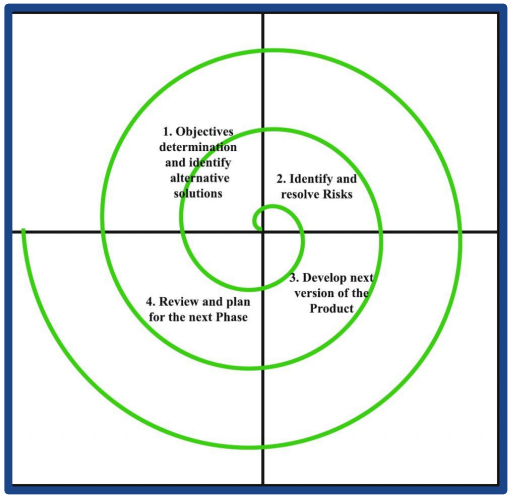

**Advantage over Waterfall**

??

# Plan & Document: Spiral

**General Idea: Prototyping + Waterfall**

- The idea is to iterate through a sequence of four phases, with each iteration resulting in a prototype that is a refinement of the previous version.
- 4 phases
  - Determine objectives and constraints of this iteration
  - Evaluate alternatives; identify and resolve risks
  - Develop and verify the prototype for this iteration
  - Plan the next iteration




**Advantage over Waterfall**

- Easier for customers to understand what they want once they see the prototype

# Plan & Document: Spiral

**Pros**

- **Risk Handling:** The projects with many unknown risks that occur as the development proceeds, in that case, Spiral Model is the best development model to follow due to the risk analysis and risk handling during each iteration.
- Good for **large and complex projects.**
- **Flexibility in Requirements:** Changes at later phase can be incorporated accurately by using this model.

# Plan & Document: Spiral

**Pros**

- **Risk Handling:** The projects with many unknown risks that occur as the development proceeds, in that case, Spiral Model is the best development model to follow due to the risk analysis and risk handling at every phase.
- Good for **large and complex projects.**
- **Flexibility in Requirements:** Changes at later phase can be incorporated accurately by using this model.

**Cons**

- **Complex and Expensive**: Spiral Model is not suitable for small projects as it is expensive.
- **Difficulty in time management:** As the number of iteration is unknown at the start of the project, so time estimation is very difficult.

# Plan & Document: Rational Unified Process

**General Idea: Spiral + Waterfall + Documentation**

- 4 phases (more dynamic)
  - **Inception**: business case, schedule and budget
  - **Elaboration**: identify use cases, design a software architecture, set the development plan, and builds an initial prototype.
  - **Construction**: code and test the product => first external release.
  - **Transition**: production, customer acceptance testing and user training

| 6 workflows / engineering disciplines (more static) |
| --- |
| Business Modeling |
| Requirements |
| Analysis and Design |
| Implementation |
| Test |
| Deployment |

**How is it different from Waterfall?**

- Each phase involves iteration

# Agile Software Development Process

**General Idea**

- Incremental refinement of prototypes with continuous feedback from the customer over the course of many 14 week iterations.
- Manage change, run compact projects with small teams, and deliver quality software on time and within budget.
- Doesn't create documentation but creates requirements as user stories as a result of frequent customer interaction
- Daily standup meetings to identify and overcome obstacles
- Often take a Kanban approach inspired by Toyota's manufacturing process

| 4 principles (vs P&D) |
| --- |
| Individuals and interactions over processes and tools |
| Working software over comprehensive documentation |
| Customer collaboration over contract negotiation |
| Responding to change over following a plan |

# Software Quality

**General Idea**

- Ensures that products under development have high quality; Creation processes and standards in an organization that lead to high quality software
- Must satisfy the needs of:
  - Customers: easy to use, doesn’t crash etc.
  - Developers: each to debug and enhance
- Terminology
  - Verification: Did you build the thing right? Met specification?
  - Validation: Did you build the right thing? Build what the customer wants?

# Software Quality

**General Idea**

- Ensures that products under development have high quality and are creating processes and standards in an organization that lead to high quality software
- Must satisfy the needs of:
  - Customers: easy to use, doesn't crash etc.
  - Developers: each to debug and enhance
- Terminology
  - Verification: Did you build the thing right? Met specification?
  - Validation: Did you build the right thing? Build what the customer wants?
- Testing
  - Problem: Non-exhaustive nature of testing

# Software Quality

**General Idea**

- Ensures that products under development have high quality and are creating processes and standards in an organization that lead to high quality software
- Must satisfy the needs of:
  - Customers: easy to use, doesn't crash etc.
  - Developers: each to debug and enhance
- Terminology
  - Verification: Did you build the thing right? Met specification?
  - Validation: Did you build the right thing? Build what the customer wants?
- Testing
  - Problem: Non-exhaustive nature of testing
  - Solution: divide into unit testing, module testing, integration testing and system / acceptance testing; Alternatives: Formal Methods

# Software Quality

**General Idea**

- Ensures that products under development have high quality and are creating processes and standards in an organization that lead to high quality software
- Must satisfy the needs of:
  - Customers: easy to use, doesn't crash etc.
  - Developers: each to debug and enhance
- Terminology
  - Verification: Did you build the thing right? Met specification?
  - Validation: Did you build the right thing? Build what the customer wants?
- Testing
  - Problem: Non-exhaustive nature of testing
  - Solution: divide into unit testing, module testing, integration testing and system / acceptance testing; Alternatives: Formal Methods

**Interesting example: Experiential Programming**

- Write tests before writing the code
- Write minimum code to pass the test => code is always tested => low chances of writing code that will be discarded
- Acceptance/System and Integration => BDD
- Unit and module => TDD

# Developer Productivity

**Clarity via conciseness**

- Small programs => Easy to evolve and understand => fewer bugs and easier to maintain
- How do programming languages do this?
  - Syntax that lets programmers express ideas using fewer characters
  - Raise level of abstraction

# Developer Productivity

**Clarity via conciseness**

- Small programs => Easy to evolve and understand => fewer bugs and easier to maintain
- How do programming languages do this?
  - Syntax that lets programmers express ideas using fewer characters
  - Raise level of abstraction

**Synthesis of Implementations**

- Automatically generated code > Manually written code
- Metaprogramming: automatically synthesize code at runtime

# Developer Productivity

**Clarity via conciseness**

- Small programs => Easy to evolve and understand => fewer bugs and easier to maintain
- How do programming languages do this?
  - Syntax that lets programmers express ideas using fewer characters
  - Raise level of abstraction

**Synthesis of Implementations**

- Automatically generated code > Manually written code
- Metaprogramming: automatically synthesize code at runtime

**Reuse**

- Reuse existing designs, procedures and functions > writing everything from scratch
- Object Oriented Programming => Inheritance
- Dynamic Typing
- Mix-ins
- Don’t have to change all copies when fixing a bug - just one place
- DRY - don’t repeat yourself

# Developer Productivity

**Clarity via conciseness**

- Small programs => Easy to evolve and understand => fewer bugs and easier to maintain
- How do programming languages do this?
  - Syntax that lets programmers express ideas using fewer characters
  - Raise level of abstraction

**Synthesis of Implementations**

- Automatically generated code > Manually written code
- Metaprogramming: automatically synthesize code at runtime

**Reuse**

- Reuse existing designs, procedures and functions > writing everything from scratch
- Object Oriented Programming => Inheritance
- Dynamic Typing
- Mix-ins
- Don’t have to change all copies when fixing a bug - just one place
- DRY - don’t repeat yourself

**Automation via tools**

- Replace tedious manual tasks with tools to save time, improve accuracy, or both.
- Concerns
  - Trade off between time to learn and time saved
  - Dependability of the tool
  - The quality of the user experience
  - How to decide which one to use
- Examples
  - Cucumber, Pivotal Tracker, RSpec, Compilers, Interpreters

## SaaS: Software as a service

- Software is run on Internet based servers that communicate among each other and allow users to access the service via a web browser
- Software as a Service (SaaS) is attractive to both customers and providers.
  - Customer: universal client (the Web browser) makes it easier for customers to use the service
  - Developer: single version of the software at a centralized site makes it easier for the provider to deliver, improve and frequent upgrade the service.

## Advantages

??

**SaaS: Software as a service**

- Software is run on Internet based servers that communicate among each other and allow users to access the service via a web browser
- Software as a Service (SaaS) is attractive to both customers and providers.
    - Customer: universal client (the Web browser) makes it easier for customers to use the service
    - Developer: single version of the software at a centralized site makes it easier for the provider to deliver, improve and frequent upgrade the service.

**Advantages**

- No local installation removes hardware compatibility requirement
- Service data is kept within service (no back up, data loss concerns)
- Easier for groups of users to collaborate on same data
- Centralized data with remote access that is good for large frequently changing data
- Developers maintain single copy of server software (instead of 1 per OS)
- Upgrades to software don’t interfere with user experience
- Encourages more competition + better features among SaaS companies

**Problem**

- When creating SaaP, developers could make extensive use of software libraries containing code to perform certain tasks common to many applications. Using these third-party libraries embodies the advantage of software reuse.

**Problem**

- When creating SaaP, developers could make extensive use of software libraries containing code to perform certain tasks common to many applications. Using these third-party libraries embodies the advantage of software reuse.

**Solution: Service oriented architecture**

- Service-oriented architecture (SOA) is a set of independent services composed together
- A service is a program that can be communicated with APIs
- SaaS service could call upon other services built and maintained by other developers for common tasks.
- Services that were highly specialized to a narrow range of tasks are called microservices
- Examples: credit card processing, search, driving directions etc..

# SOA

**Reusable**: can combine existing services where each microservice is implemented with their own language/framework (implementation hidden behind API)

**Performance:** for each microservice, need to dig into the software stack of a network interface due to layers of APIs, which can cause performance penalty: State is split across services

# SOA

| | |
|---|---|
| **Reusable**: can combine existing services where each microservice is implemented with their own language/framework (implementation hidden behind API) | **Performance:** for each microservice, need to dig into the software stack of a network interface due to layers of APIs, which can cause performance penalty: State is split across services |
| **Easy testing:** each microservice does only one thing, testing that one thing in isolation is easier<br>Allows developers to use best tool for job for each microservice | **Dependability:** Partial Failure is possible in SOA since microservices can fail independently from others, so it is not just a measure of “app is working” vs “app isn’t working”.In cases like this testing can be harder (integration and system tests, validation tests) |

# SOA

| | |
|---|---|
| **Reusable**: can combine existing services where each microservice is implemented with their own language/framework (implementation hidden behind API) | **Performance:** for each microservice, need to dig into the software stack of a network interface due to layers of APIs, which can cause performance penalty: State is split across services |
| **Easy testing:** each microservice does only one thing, testing that one thing in isolation is easier<br>Allows developers to use best tool for job for each microservice | **Dependability:** Partial Failure is possible in SOA since microservices can fail independently from others, so it is not just a measure of “app is working” vs “app isn’t working”.In cases like this testing can be harder (integration and system tests, validation tests) |
| **Agile Friendly**: works best with small-medium teams, since SOA allows small teams to develop microservices and then combining them. | **Development Work:** each microservice has to build their own interface instead of a single monster interface for the entire application, REST simplifies this each microservice essentially being end-to-end services that can stand on their own |

# SOA

| | |
|---|---|
| **Reusable**: can combine existing services where each microservice is implemented with their own language/framework (implementation hidden behind API) | **Performance:** for each microservice, need to dig into the software stack of a network interface due to layers of APIs, which can cause performance penalty: State is split across services |
| **Easy testing:** each microservice does only one thing, testing that one thing in isolation is easier<br>Allows developers to use best tool for job for each microservice | **Dependability:** Partial Failure is possible in SOA since microservices can fail independently from others, so it is not just a measure of “app is working” vs “app isn’t working”.In cases like this testing can be harder (integration and system tests, validation tests) |
| **Agile Friendly**: works best with small-medium teams, since SOA allows small teams to develop microservices and then combining them. | **Development Work:** each microservice has to build their own interface instead of a single monster interface for the entire application, REST simplifies this each microservice essentially being end-to-end services that can stand on their own |
| Same team is responsible for developing, testing and operating the microservice so **customer-feedback cycles** can be done quickly and efficiently | Dev/Ops: developers need to know about operations to be able to manage functionality and performance. If **“you build you run”** then you need to know ops deeper |

Cloud computing provides a scalable and dependable hardware computation and storage for SaaS

**Need of SaaS**

- Communication: allow any customer interact with the service
- Scalability: central facility running the service must deal with the fluctuations in demand during the day and during popular times of the year for that service as well as a way for new services to add users rapidly.
- Availability: both the service and the communication vehicle must be continuously available

**How can these needs be fulfilled?**

# Cloud Computing

Cloud computing provides a scalable and dependable hardware computation and storage for SaaS

**Need of SaaS**

- Communication: allow any customer interact with the service
- Scalability: central facility running the service must deal with the fluctuations in demand during the day and during popular times of the year for that service as well as a way for new services to add users rapidly.
- Availability: both the service and the communication vehicle must be continuously available

**How can these needs be fulfilled?**

- **Cluster**: small scale computers connected by Ethernet switches > expensive large scale computer
  - More scalable than conventional servers: due to reliability on Ethernet switches
- **Virtual machine**: software that imitates a real computer. An OS can be run on top of the VM abstraction
  - Can imitate with low overhead as well as simplify software distribution within a cluster
  - multiple apps can share hardware with each app believing that it has its own copy of the OS
  - OS-level virtualization: Docker

# Cloud Computing

Cloud computing provides a scalable and dependable hardware computation and storage for SaaS

**Need of SaaS**

- Communication: allow any customer interact with the service
- Scalability: central facility running the service must deal with the fluctuations in demand during the day and during popular times of the year for that service as well as a way for new services to add users rapidly.
- Availability: both the service and the communication vehicle must be continuously available

**How can these needs be fulfilled?**

- **Cluster**: small scale computers connected by Ethernet switches > expensive large scale computer
  - More scalable than conventional servers: due to reliability on Ethernet switches
- **Virtual machine**: software that imitates a real computer. An OS can be run on top of the VM abstraction
  - Can imitate with low overhead as well as simplify software distribution within a cluster
  - multiple apps can share hardware with each app believing that it has its own copy of the OS
  - OS-level virtualization: the apps are also able to share the operating system. Tool: Docker

- Cheap: Clusters are 20x cheaper than expensive large scale computers
- Dependability via extensive use of redundancy in hardware and software.

# Public Cloud Services

**General Idea**

- Public cloud services, utility computing, or often simply cloud computing offers computing, storage, and communication at pennies per hour

**Economies of Scale**

- They take advantage of economies of scale
  - Many companies share large data centers
  - Smaller data centers often run at only 10% to 20% utilization.
  - These companies profit by offering there datacenter hardware on a pay-as-you-go basis
    - low-cost utility computing
    - Acquire resources immediately as your customer demand grows and releasing them immediately when it drops

# Legacy Code

Successful software can live decades and is expected to evolve and improve

**Terminology**

- Beautiful Code: long-lasting code that is easy to evolve.
- Legacy Code: software that, despite its old age, continues to be used because it meets customers' needs. However, its design or implementation may be outdated or poorly understood.
  - 60% Maintenance Costs ⇒ Adding new functionality to legacy code
  - 17% Maintenance Costs ⇒ Bug Fixing
- Unexpectedly Short-lived Code: worst case, code that is soon discarded because it doesn't meet customers' needs.

# Ruby Review

# Ruby

- Dynamically typed programming language
  - No need to declare a type => Like python
  - `x = "hi there"  #String`
  - `x = 3  #FixNum or Integer`
  - `x = 3.5  #Float`
  - `x = 3..10  #Range - inclusive`
  - `x = 3...10  #Range - exclusive`

- Object-oriented
- Interpreted + Just-in-time compiler
- snake_case variable names
- Titlecase (or first letter is uppercase) for constants
- Print statement equivalent => puts “hello world”
- Comments
  - Single line: `#`
  - Multiline
    - ```
      =begin
      ...
      =end
      ```
      ```

# Ruby Objects

- No primitive data types. Everything is an object
  - Common Data Types (Classes): Numbers, Strings, Ranges, nil (null in Java)

- Uses "dot-notation" like Java objects for instance variables and methods
- You can find the class of any variable
  - x = "hello" x.class → String
- Different Classes of Numbers – FixNum, Float
  - 3.eql?2 False
  - -42.abs 42
  - 3.4.round 3
  - 3.6.round 4
  - 3.zero? False

- Changing types of objects
  - to_i – converts to an integer (Fixnum)
  - to_f – converts a String to a Float
  - to_s – converts a number to a String

# Ruby Arrays

- Zero indexed. Negative index values can be used.
- Uses square brackets, []
- Elements are separated by commas
- Store an assortment of types within the same array
  - a = [1, 4.3, "hello", 3..7]
- You can assign values to an array at a particular index
- Arrays increase in size if an index is specified out of bounds and fill gaps with nil

**Example**

```
a = [1, 4.3, "hello", 3..7]
a[4] = "goodbye"
a
> [1, 4.3, "hello", 3..7, "goodbye"]
a[6] = "hola"
a
> [1, 4.3, "hello", 3..7, "goodbye", nil, "hola"]
a[-1]
> “hola”
```

# Ruby Conditionals

```
if age < 35
    puts "young whipper-snapper"
elsif age < 105
    puts "80 is the new 30!"
else
    puts "wow..."
end
```

```
puts "young whipper-snapper" if age < 35      #inline
```

- Unless is the logical opposite of if

# Ruby Loops

```
array = [1, 2, 3, 4, 5]
```

```
array.each do |x|
        puts x
end


array.each { |x|
        puts x
}
```

```
for x in array
        puts x
end


4.times do
        puts “hello world”
end
```

```
while i < 5
        puts i
        i += 1
end


loop do
        x += 1
        break if x > 1
end
```

- Cannot use "i++", "--i" but can use "i += 1", "i -= 1"
- Until is the logical opposite of while

# Ruby Methods

```
        def method_name(parameter1, parameter2, ...)
            statement
        end
method_name(arg1, arg2)
```

```
        def method_name(num1, num2, ...)
            return num1 + num2
        end
        def method_name(num1, num2, ...)
            num1 + num2
        end
```

# Ruby Blocks

- Blocks of code that can be passes, called or yield
- Defined by curly braces {} or do/end statement

```
{|v| puts v}

do |v|

    puts v

end


(0..2).each {|v| puts v} => 012
```

# Ruby Yield

- yield statements goes hand-in-hand with blocks
- The code of a block is executed when a yield statement is called

```
def yielding
      puts "the program is executing the code inside the method"
      yield
      puts "now we are back in the method"
end

yielding { puts "the method has yielded to the block!" }
```

```
the program is executing the code inside the method

the method has yielded to the block!

now we are back in the method
```

# Ruby Iterator

- Collection-like objects like arrays and range have iterator methods
- Iterator takes a block as callback and yield the block with each element in the collection

```
[1, 2, 3].each do |n|

  text = "Current number is: #{n}"

  puts text

End
```

```
Current number is: 1

Current number is: 2

Current number is: 3
```

# Ruby Class

```
class Customer
    $acquisition_cost = 0
    @@cust_purchase_limit = 10

    def initialize(id, name, addr)
      @cust_id = id
      @cust_name = name
      @cust_addr = addr
   end

    def buy(product)
        puts product
    end
end

c1 = Customer.new(1, “Kyle”, “1234 Shattuck Avenue, Berkeley CA”)
c1.buy(“ball”) => “ball”
C1.name = “kyle”
```

# Ruby Operator Precedence

| Method | Operator | Description |
|---|---|---|
| Yes | `[ ]` `[ ]=` | Element reference, element set |
| Yes | `**` | Exponentiation (raise to the power) |
| Yes | `!` `~` `+` `-` | Not, complement, unary plus and minus (method names for the last two are `+@` and `-@`) |
| Yes | `*` `/` `%` | Multiply, divide, and modulo |
| Yes | `+` `-` | Addition and subtraction |
| Yes | `>>` `<<` | Right and left bitwise shift |
| Yes | `&` | Bitwise `AND' |
| Yes | `^` `\|` | Bitwise exclusive `OR' and regular `OR' |
| Yes | `<=` `<` `>` `>=` | Comparison operators |
| Yes | `<=>` `==` `===` `!=` `=~` `!~` | Equality and pattern match operators ( `!=` and `!~` may not be defined as methods) |
| | `&&` | Logical `AND' |
| | `\|\|` | Logical `AND' |
| | `..` `...` | Range (inclusive and exclusive) |
| | `? :` | Ternary if-then-else |
| | `=` `%=` `{` `/=` `-=` `+=` `\|=` `&=` `>>=` `<<=` `*=` `&&=` `\|\|=` `**=` | Assignment |
| | `defined?` | Check if specified symbol defined |
| | `not` | Logical negation |
| | `or` `and` | Logical composition |
| | `if` `unless` `while` `until` | Expression modifiers |
| | `begin/end` | Block expression |

# Attendance Form Link

https://tinyurl.com/discussion-quiz-1

# Let’s Practice!

# Quiz Practice #1

Which productivity mechanism does service-oriented architecture (SOA) best exemplify?

- Clarity via Conciseness
- Synthesis
- Reuse
- Automation via Tools

# Quiz Practice #1

Which productivity mechanism does service-oriented architecture (SOA) best exemplify?

- Clarity via Conciseness
- Synthesis
- Reuse
- Automation via Tools

# Quiz Practice #2

Which of the following functionality would most likely be defined as a class method?

- return the total number of Movie objects created
- update the ratings of a Movie object
- compare and sort the actors of the Movie object in an alphabetical order

# Quiz Practice #2

Which of the following functionality would most likely be defined as a class method?

- return the total number of Movie objects created
- update the ratings of a Movie object
- compare and sort the actors of the Movie object in an alphabetical order

# Quiz Practice #3

(Select all that apply.) Which statements comparing Plan-and-Document (P&D) with Agile software engineering processes are true?

- The basic types of activities involved in software engineering are the same in P&D and Agile methodologies
- Because Agile tends to focus on small teams, it cannot be used effectively to build large systems.
- The Waterfall methodology involves the customer much more heavily at the beginning and end of the life cycle, whereas the XP (extreme programming) methodology involves the customer throughout the lifecycle.
- The Spiral methodology combines elements of the waterfall model with intermediate prototypes.

# Quiz Practice #3

(Select all that apply.) Which statements comparing Plan-and-Document (P&D) with Agile software engineering processes are true?

- The basic types of activities involved in software engineering are the same in P&D and Agile methodologies
- Because Agile tends to focus on small teams, it cannot be used effectively to build large systems.
- The Waterfall methodology involves the customer much more heavily at the beginning and end of the life cycle, whereas the XP (extreme programming) methodology involves the customer throughout the lifecycle.
- The Spiral methodology combines elements of the waterfall model with intermediate prototypes.

# Quiz Practice #4

Internal data centers could get the same cost savings as Warehouse Scale Computers (WSCs) if they embraced SOA and purchased the same type of hardware.

- True
- False

# Quiz Practice #4

Internal data centers could get the same cost savings as Warehouse Scale Computers (WSCs) if they embraced SOA and purchased the same type of hardware.

- True
- False

# Quiz Practice #5

```
def boolean_potpourri
 first = true and false
 second = true && false
 third = (true and false)
 puts first, second, third
end
```

Determine what the following code snippet would print out:

- true, true, false
- true, false, false
- false, SyntaxError (Unexpected Identifier)
- false, false, false

# Quiz Practice #5

```
def boolean_potpourri
 first = true and false
 second = true && false
 third = (true and false)
 puts first, second, third
end
```

Determine what the following code snippet would print out:

- true, true, false
- true, false, false
- false, SyntaxError (Unexpected Identifier)
- false, false, false

# Quiz Practice #6 - Open ended

Create a situation where the following development processes must be used and give three reasons why it should be used.

- Waterfall
- Spiral
- Rational Unified Process
- Agile

# More Quiz Practice

1. What are the differences between Agile and Waterfall?
2. Can you design and build hardware using the Agile Process?
3. What are some formal methods of verification and what are there pros and cons?
4. How does SaaS differ from traditional software? What makes it more advantageous?