→ What makes a programmer pragmatic?

- → Early Adopter
- → Inquisitive → "Ask questions"
- → Critical Thinker
- → Realistic
- → Jack of all trades

① Care about your craft.

② Think! About Your Work.

→ Old IBM corporate Motto: Constantly Think, Critique your work
→ Sounds like Hard Work → Realistic
→ Actively involved with a job you love.
→ Write code that is easier to read
⇒ less time wasted in meeting :)

Individual Pragmatists, Large Teams

→ Some people think there is no room for individuality on large teams.

Above statement is not true, since individuality matters a lot!

It's a continuous Process!

Topic 1 : It's Your Life

→ People complain about various issues, main answer to that is "why can't you change it!"

Software Dev. is closest where you have control.

③ You Have Agency

Industry offers set of opportunity. Be proactive and take them.

⇓ you will have to work for it.

Topic 2 : The Cat Ate My Source Code

⇒ Take charge of your carrier, and don't be afraid to admit ignorance or error.

⇒ Be honest and direct and deal with things profesionally.

TEAM TRUST

→ trust
→ reliability

TAKE RESPONSIBILITY

let's say → you took ↑ but you don't have **full control.**

↳ Assess Risk and

don't take responsibility

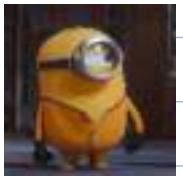
④ 'Provide options, Don't make lame excuses. or blame

Before approaching: Why something can't be done?
when they ask "did you try this/that"
↳ how to handle this

Don't say can't be done	→	Provide options
I don't know	→	I will find out

Topic 3: Software Entropy
↳ "disorder"
↳ goes to max.

of Software
"disorder" → max } Software Rot
"Technical Debt"



Implied notion that they will pay someday

⑤ Don't live with Broken Windows.
→ Bad Design
→ Wrong Decision

→ Poor Code

Psychology and Culture → Matters!

first, DO NO HARM!

Topic 4: Stone soup and Boiled Eggs

⑥ Be the catalyst for Change!

There may be "start-up fatigue"

⑦ Remember the Big Picture

Topic 5: Good Enough Software

Writing a perfect software → impossible

→ focus on writing a good enough, good enough for users., future maintainers, for your peace of mind.

"Good Enough" should not be taken as bad, inefficient code that works

⑧ Make quality a requirement.

Topic 6: Your Knowledge Portfolio

Your knowledge and experience → "assets" — expiring!

Managing an knowledge portfolio is not that different from financial.

5 Tips:

1. Invest Regularly – Small but daily – habit
2. Diversify : Ins/Out of your work + other stuff
3. Manage Risk : Don't put all your eggs in one basket.
4. Buy low, sell high : Java early adopter example.
5. Review and Rebalance :

⑨ Invest Regularly in Your Knowledge Portfolio.

Goals:
1. Learn at least 1 language every year.
2. Read a technical book every month.
3. Read non-technical books too.
4. Take classes
5. Participate in local user groups & meetups.
6. Experiment with different environments.
7. Stay Current!

⇒ Even if you won't use that knowledge in resume, or your projects.

⇒ Learn to become Knowledgeable! Even if your

project doesn't use that tech, but it can use those ideas.

> Cross Pollination of Ideas is important

Opportunities for learning, → If someone ask something and you don't have faintest idea → "ask others" → communication.
↓
Build Networks !

⑩ Critically Analyze what you read and hear

How to critically think ?

1. Ask the five Why's
2. Who does this benefit.
3. What's the context.
4. When or where this should work
5. Why is this problem ?

Topic 7 : Communicate

Good Idea is an "orphan" without effective communication

⑪ English is Just Another Programming Language.

Some Important Ideas Regarding Communication.

→ Know your Audience.

- way of communication is very important
- gather feedback!

→ Know what you want to say

- write outline.
- refine

→ Choose your Moment

→ Choose your Style.

→ Make it look good.

- Ideas are important.
  - → deserve a good looking vehicle.

→ Involve your Audience.

- take early review/feedback
- improves communication.

→ Get Back to People!

- always respond! → "say u b back"

(12) Its Both What you say and the way you say

→ Documentation.

(13) Build Documentation In, Don't Bolt It On

Topic 8: The Essense of Good Design.

(14) Good Design is Easier to change than Bad Design

ETC: Easy to Change.

→ ETC is a value, not a Rule!
It should be at back of mind.

→ There is an implicit premise in ETC. Person is able to tell which of many paths will be easier to change in the future, since it may not be obvious.

There are 2 ways to deal with it.

1. Since we are not sure what form your change will take, fall back to ETC path.
2. Treat this as a way to develop instincts. Leave some marker or comments in code.

Topic 9: DRY - The evils of duplication.

→ Programers are always in maintenance mode. Requirement, understanding and clients change all the time.

Only way to develop software reliabely, and to make our development easier to understand and maintain, is to follow what we call DRY principle.

(15) DRY – Don't Repeat Yourself.

"Every piece of knowledge must have a single, unambigous, authoritative representation within a system".

DRY Is More than code.

(16) Make it Easy to Reuse.

Topic 10: Orthogonality

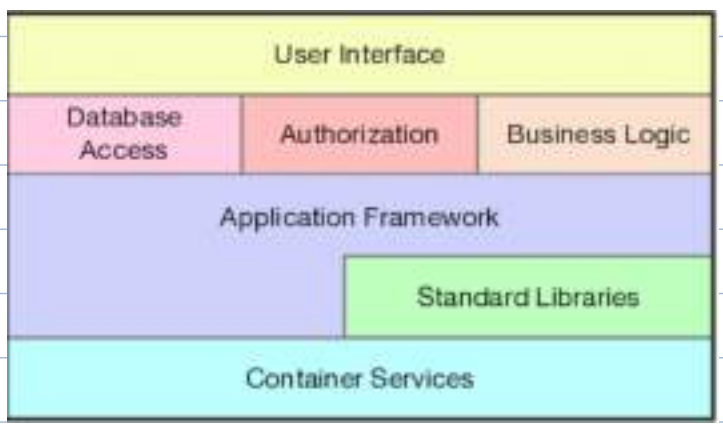
In computing, it signifies independence or decoupling.

Benefits of Orthogonality

(17) Eliminates Effects between unrelated things.

We get 2 Benefits: Increased productivity, reduced risk.

Design



System should be composed of a set of co-operating modules, each of which implements independent functionality.

Sometimes these components maybe organised in layer, each providing a lvl of abstraction.

Coding

→ Keep your code decoupled.
→ Avoid global data.
→ Avoid similar function.

Testing

Unit Testing becomes easier in orthogonal system

Documentation

Axes : Content and Presentation.
Write MD and leave presentation to some tool

Topic 11 : Reversibility

(18) There are no final decisions.

Flexible Architecture

Considering architectural volatility in these times. Best course of action is that, make it easy to change your code. Hide third party APIs behind your abstraction layers. Break your

code in components even if deployed as monolith.

(19) fargo following fads.

Topic 12: Tracer Bullets

Code that glows in the dark.

(20) Use tracer Bullet to find the target.

Look for requirements → that define system,
Look for areas of doubts, where biggest risk is?
Prioritize development accordingly.

Advantages of the tracer code approach.

- → User get to see something working early
- → Developer build a structure to work in
- → You have integration platform.
- → You have something to demonstrate.
- → You have better feel for progress.

→ Tracer Bullets Don't Always hit their target.

→ Tracer Code (looks at how things fit together) V/S Prototyping. (aims to explore)

code ... as a whole

→ tracer code is lean but complete and forms skeleton of project.

specific aspects of ... systems.

→ produces disposable code

Topic 13 : Prototype and Post-it Notes.

Things to Prototype

→ Architecture
→ New functionality in an existing system
→ Structure or Contents of external data.
→ third party tools or components.
→ performance issues
→ user interface design.

(21) Prototype to Learn.

When building prototype, details that can be ignored

→ Correctness
→ Completeness
→ Robustness
→ Style

Prototyping Architecture

Specific areas to look for

→ Are the responsibility of the major areas well

defined and appropriate

→ Are the collaboration between major components well defined

→ Is coupling minimized?

→ Can you identify potential sources of duplication

→ Are interface definition & constraints acceptable

How Not to use Prototype!

Make sure everyone knows that you are writing disposable code.

Topic 14: Domain Languages.

(22) Program close to the Problem Domain.

Tradeoff's between Internal and External language

→ take advantage of features of its host

→ downside: youre bound to syntax & semantics of ruby

Topic 15: Estimating

(23) Estimate to avoid supprises.

▶ How Accurate is Accurate Enough?

- Where do estimates come from
  - Understand what's being asked.
  - Build a model of system
  - Break model into components
  - Give each parameter a value
  - Calculate the answer
  - Keep track of your estimation prowess

→ Estimating Project Schedule

* Painting the Missile.

Not one hard number but range of scenario

PERT technique

* Eating the Elephant

→ Check Requirements
→ Analyse Risk (prioritize riskies items)
→ Design, Implement, Integrate
→ Validate with users.

(24) Iterate the schedule with code.

What to say when asked for an Estimate

"I'll get back to you" :)

Topic 16 : The power of plain text.

(25) Keep Knowledge in Plain Text

Powers of plain text

→ Insurance against obsolescence
→ Leverage Existing tools
→ Easier Testing.

Topic 17 : Shell Games

(26) Use the power of command shells

A shell of your own
→ Set color themes
→ Configure a prompt
→ Alias and shell function
→ Command completion

Topic 18 : Power Editing

(27) Achieve Editor fluency

Topic 19 : Version Control

(28) Always use version control.

→ Branching Out
- Take backup of your user preferences, dotfiles, homebrew installs, Ansible scripts all current projects.

→ Version Control as Project Hub.

Topic 20 : Debugging

Embrace the fact that debugging is just problem solving.

(29) fix the Problem, Not the Blame

Debugging Mindset

(30) Don't Panic

Debugging Strategies

→ Reproducing Bugs

(31) failing Test Before fixing code

→ Coder in Strange Land

(32) Read the Damn Error Message.

→ Bad Result

→ Sensitivity to Input Values

→ Regression across Releases

The Binary Chop : Write test that fails current release. then do binary search across releases.

→ Logging or Tracing

→ Rubber Ducking

→ Process of Elimination

(33) Select isn't Broken!

The Element of Suprise

(34) Don't Assume It - Prove it

Topic 21 : Text Manipulation

(35) Learn a Text Manipulation Language.

sed, awk, gawk, perl

Topic 22 : Engineering Daybooks

3 Benefits

1. More Reliable than Memory
2. gives place to store ideas that aren't immediately relevant to task at hand.
3. Acts kind of like Rubber Duck.

Chapter 4 : Pragmatic Paranoia

(36) You can't write Perfect Software

Build defenses against your own mistake.

Topic 24: Dead programs tell No Lies.

→ Rather than ignoring errors, try to find it and Read the damn error message

→ Catch and Release is for fish.

(38) Crash Early

Topic 25: Assertive Programming

(39) Use assertion to Prevent Impossible

Assertion and Side Effects.
→ avoid problems like Heisenbug: debugging that changes behaviour of system being debugged.

Leave assertions Turned on.

Topic 26: How to Balance Resources.

(40) finish what you start
→ file opening/closing and coupled fn's

(41) Act Locally.
when in doubt always reduce scope.

Nest Allocation

⇒ Objects and Exception (Constructor / Deconstructor)
⇒ Balancing and Exception

1. Variable Scope
2. Use finally in a try.... catch block

⇒ An exception antipattern

Correct pattern → thing = allocate()
begin
process(thing)
finally
deallocate(thing)
end.

→ When you can't Balance Resources
→ Checking the Balance.

Topic 27: Don't Outrun your headlights.

(42) Take small step — always checking for feedback and adjusting before proceeding.

feeback: •> Results in a REPL provide feedback
•> Unit test provide feedback on your last code change
•> User demo and conversation.

(43) Avoid fortune telling

Chapter 5: Bend or Break.

Topic 28: Decoupling

Coupling is enemy of change, because it links together things that must change in parallel.

(44) Decoupled code is easier to change

3 Concepts

1. Train Wrecks

(45) Tell, Don't Ask
Don't make decisions based on internal state of an object and then update the object.

The Law of Demeter: Written by Ian Holland

LoD says that method defined in class C should only call
- → other instance method in C
- → Its parameters
- → Methods in objects that creates, both on the stack and in the heap.
- → Global variables. } Not so good

(46) Don't Chain Method Calls.

The Evils of Globalisation

→ Insidious source of coupling

(47) Avoid Global Data.

(48) If it's Important Enough to be global, Wrap it up in an API.

Inheritance adds coupling.

Topic 29: Juggling the Real World.

Writing responsive Application

EVENTS → represents availability of information.

four strategies to help in case an event triggers
1. finite state machines
2. The Observer pattern
3. Publish / Subscribe
4. Reactive Programming and streams (RxJS)

Topic 30: Transforming Programming

(49) Programming is About Coding. But Programs are about Data.

- Finding Transformation
- Keep on Transforming
- Putting it all together.

(50) Don't hoard state, pass it around.

Topic 31 : Inheritance Tax

Problems using Inheritance
→ Inheritance is coupling.

(51) Don't pay inheritance tax!

Alternatives

→ Interfaces and protocols
→ Delegation
→ Mixins and traits.

(52) Prefer Interfaces to Express Polymorphism.

(53) Delegate to Services: Has-A Trumps Is-A

(54) Use mixins to share functionality.

Topic 32 : Configuration

(56) Parameterize your app Using External Config.

Common data to be put in configuration
→ Credential for external services
→ Logging levels and destination
→ port, IP Address, Machine, and cluster name the app uses.
→ Environment specific validation parameter
→ externally set values.
→ Site-specific formatting details
→ License keys

Static Configuration → yaml, json.

Configuration as a service.

Don't write Dodo Code.

Chapter 6: Concurrency

Context-switch / Timesharing

Concurrency is when the execution of two or more pieces of code act as if they run at the same time.

Parallelism is when they do run at the same time

Topic 33: Breaking Temporal Coupling

Coupling in Time (why?) → We are taught to think of code in sequential manner.

Looking for Concurrency

(56) Analyze Workflow to Improve Concurrency

Activity Diagrams helps us tackle dependencies and helps us chart course for parallelism or concurrency.

Topic 34: Shared State is Incorrect State

(57) Shared State is Incorrect State.

Non atomic updates : Classic example where two threads try to update (cnt = cn + 1)

↑ actually 3 ops

To make ops atomic use semaphores and other forms of mutual exclusion.

We give semaphore to give control to thread but what happens if thread doesn't give back control ?

→ Make the resource transactional
→ Multiple resource transaction.
→ Non-Transactional Updates.

(58) Random failures are often Concurrency Issues

Other Kinds of Exclusive Acces.

Many libraries have support for some kind of exclusive access to shared resources. They call it mutex (mutually exclusive), monitors or semaphores.

Topic 35 : Actors and Processes

→ Independent Virtual Processor with its own local state. Each actor has a mailbox which kicks into action as soon as a message arrives or else goes back to sleep.

→ process : general purpose virtual processor, often implemented by OS to facilitate concurrency.

Actors can only be concurrent.

(59) Use actors for Concurrency without shared state.

few things not in definition of actor

→ There is no single thing thats in control
→ only state in system is held in menage and in local state of each actor.
→ All messages are one way – there's no concept of reply.
→ An actor processes each message to completion, and only proceses one at a time.

No Explicit Concurrency in case of Actors.

Erlang sets the stage. → based one some what similar lightweight proceses.

Topic 36 : Blackboards

Computer based blackboard systems were originally used in AI application where problem to be solved were large and complex.

One of the first of its kind was David Gelernter's Linda.

Later came distributed blackboard systems like Javaspace and Tspace.

(60) Use Blackboard to Co-ordinate Workflow

Chapter - 7 : While you are Coding

Topic 37 : Listen to your Lizard Brain.

(61) Listen to your lizard brain

insticts, non-conscious brain.

Take break if feeling stuck at some problem. make doodles of problem, explain it to your Co-worker or your rubber ducky.

It's Playtime

Author has stared at empty buffers for long and devised a way.
He tells himself that he is making a proto-type only. Maybe you are using a new framework and want to know how data

binding works. Explore new ideas, algorithms

Not just your code: try to learn from others code.

Topic 38: Programming by Coincidence.

Buoyed by false confidence, we may charge ahead in oblivion. So should we rely on coincidences do we?

Sometimes we might. Sometimes its pretty easy to confuse a happy coincidence with a purposeful plan.

Accidents of Implementation → things that happen simply because that's the way code is written.
You end up relying on undocumented error or boundry condition.

Close Enough Isn't → example of hardware units collecting data but incorrect processing of time zone giving ±1 delta and programmer just started agreeing +1, -1 is just calculation error.

Phanthom Patterns → Humans can find patterns even when there are none.
Don't assume, prove it.

Accident of Context →

(62) Don't program by coincidence.

How to program deliberately

→ always be aware of what you are doing
→ can you explain code in detail to a junior programmer
→ don't code in dark. Build an application you don't fully grasp, you will be bitten by coincidences
→ Proceed from a plan
→ Rely on reliable things. Don't depend on assumption
→ Document by assumption
→ Don't just test your code, test your assumption as well.
→ prioritize your effort
→ Don't be slave to history.

Topic 39: Algorithm speed

What do we mean by Estimating Algorithm?
Utilise Big-O notation

$$O\left(\frac{n^2}{2}+3n\right) \approx O\left(\frac{n^2}{2}\right) \approx O\left(n^2\right)$$

Common Sense Estimation

→ Simple loops : O(n)
→ Nested loops : O(mxn)
→ Binary Chop : O(lgn)
→ Divide and Conq : O(n lgn)
→ Combinatoric : O(n!)

(63) Estimate the Order of your algorithm

(64) Test Your Estimates

Best Isn't Always Best : Be wary of premature optimization.

Topic 40 : Refactoring

Code needs to evolve ; its not static.

Refactoring is disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour.

Critical parts of above defⁿ

① Activity is disciplined

② external behaviour doesn't change

When should you Refactor

→ Duplication
→ Non-Orthogonal design
→ Outdated Knowledge

→ Usage
→ Performance
→ The Test Pass

(65) Refactor Early and Refactor Often

How do you refactor ?

→ Don't try to refactor and add functionality at the same time.
→ Make sure you have good tests before you begin refactoring. Run these tests often.
→ Take small, deliberate steps : move from one class to another, split a method, rename a variable.

Topic 47: Test to code

(66) Testing is not about finding Bugs.

Tests drive Coding

(67) A Test is the first User of Your Code.

Tightly coupled code is hard to test!

Test driven Development (TDD)

Basic cycle of TDD is:

1. Decide on a small piece of functionality you want to add
2. Write a test that will pass once that functionality is implemented.
3. Run all tests. Verify that the only failure is only the test you just wrote.
4. Write the smallest amount of code needed to get the test to pass, and verify that tests now run cleanly.
5. Refactor your code.

Idea is this cycle should be short! a matter of minutes.

Drawbacks of Test Driven Development

→ Spending lots of time to get 100% coverage.
→ lots of redundant tests.
→ designs are mostly start at bottom and work their way up.

(68) Build End-to-End, Not Top-Down or Bottom Up

TDD: You need to know where you are going.

Back to Code: Component-based development.

Unit Testing

Testing against contracts.

(69) Design to Test.

Ad Hoc Testing
Build a Test Window
A Culture of Testing

(70) Test you software, or Your Users will.!

Topic 42: Property Based Testing

(71) Use property Based Tests to Validate Assumptions

In python try out Hypothesis. → generates test input

Finding Bad Assumption

→ Property Based Tests often Surprise You.
→ Property Based Tests also help your design.

Topic 43: Stay Safe Out There

The Other 90%

Security Basic Principles

1. Minimize attack surface area

2. Principle of Least Preivledge
3. Secure defaults
4. Encrypt Sensitive Data
5. Maintain Security Updates

(72) Keep It Simple and minimize attack surface

(73) Apply Security patches Quickly

Common Sense v/s Crypto

first and most important rule when it comes to crypto: "never do it yourself".

Topic 44: Naming Things

→ Honour the local culture

→ Consistency

→ Renaming is Even harder

(74) Name Well: Rename when Needed.

Topic 45: The Requirement Pit

Many books and tutorials refer to "gathering"

requirements which implies that they already exists waiting to be found. Often we are far away from them and they may not even exist.

(75) No one knows Exactly what they want.

The Requirements Myth

(76) Programmers help people understand what they want.

Programming as Therapy

You role is to interpret what client says and to feed back to them the implication

Requirements are Procces

(77) Requirements are learned in a feedback loop.

Walk in Your Clien't Shoes

(78) Work with a user to think like a User.

Requirements v/s Policy

(79) Policy IS Metadata.

Requirements v/s Reality

client says " yes, it does what I want, but not how I want ".

Documenting Requirements.

Best documentation → code.

can't deliever to client.

Requirement Documents are not for clients.

Requirement Documents are for planning.

→ User Stories → Status and priority.

→ Overspecification → dangerous.

Requirements != Design, Architecture, UI

→ need !

(80) Use a project glossary.

Topic 46 : Solving Impossible Puzzle.

Secret to solve a puzzle : Identify real (not imagined) constraints, and find a solution.

Some constraints are absolute, while others are just percieved notions.

Degree of freedom.

(81) Don't Think outside Box - find the box.

Get out of your own way.
fortune favors the prepared mind.

Topic 47 : Working Together.

Pair Programming → one of practices of eXtreme Programming.

Mob Programming

What should I Do? Try both stratigies and see what suits you.

(82) Don't Go into the Code Alone.

Topic 48 : The Essense of Agility

(83) Agile is not a noun : Agile is how you do things.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** *over processes and tools*
- **Working software** *over comprehensive documentation*
- **Customer collaboration** *over contract negotiation*
- **Responding to change** *over following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

There can never be an Agile Process. Agility is all about responding to change, unknowns that you encounter after you set out.

So what do we do?

1. Work out where you are.
2. Make the smallest meaningful step towards where you want to be
3. Evaluate where you end up, an fix anything you broke.

Topic 49 : Pragmatic Teams

A Team is a small, mostly stable entity of its own. Fifty people are not team, they are horde.

(84) Maintain Small, Stable Teams

Some concepts which are still relevant in context of teams.

→ No Broken Windows
→ Boiled Frogs

→ Schedule your knowledge portfolio.

A team work on more than feature, e.g.
- => old systems Maintenance
- => Process Reflection and Refinement
- => New tech experiments.

-> Learnings and still improvements.

(85) Schedule it to make it happen

→ Communicate Team Presence
→ Don't repeat yourselves
→ Team tracer Bullets

(86) Organize fully functional teams

→ Automation
→ Know when to stop adding paint.

Topic 50: Coconuts don't cut it

Context Matters

(87) Do what works, Not what's fashionable.

One Size fits no one well

Take best pieces from any particular methodology and adapt them for use.

The Real Goal

The goal ofcourse isn't to "do Scrum", "do agile", "do lean" or whatever. The goal is to be in position to deliever working software that gives the user some new capability at moments notice.

(88) Deliever when Users need it.

Once infra is in order, you can decide to work.

Beginners → Scrum for project management
extreme programing (XP) for technical practices.

Experienced → Kanban and Lean Techniques.

Topic 51 : Pragmatic Starter Kit

3 leg support of Every Project
1. VCS
2. Regression Testing
3. Full Automation

(89) Use Version Control to Drive Builds, Tests, and Releases

(90) Test Early, Test Often, Test Automatically

(91) Coding ain't done till all the tests run.

→ Unit Testing
→ Integration Testing
→ Validation and Verification
→ Performance Testing
→ Testing the tests
(92) Use Saboteurs to test your testing
Netflix Chaos Monkey.
→ Testing Thoroughly

(93) Test State Coverage, not code coverage

→ Property Based Testing.

(94) Find Bugs Once

Full Automation

(95) Don't Use Manual Procedures.

Topic 52: Delight Your Users

How will you know that we've all been successful a month (or a year, or whatever) after this project is done?

You may well be surprised by the answer. A project to improve product recommendations might actually be judged in terms of customer retention; a project to consolidate two databases might be judged in terms of data quality, or it might be about cost savings. But it's these expectations of business value that really count—not just the software project itself. The software is only a means to these ends.

Topic 53: Pride and Prejudice

(97) Sign Your Work

Your signature should come to be recognised as an indicator of quality.

The Moral Compass

1. Have I protected the user?
2. Would I use this myself.

(98) Do no harm

(99) Don't enable scumbags

(100) It's your Life.
Share it. Celebrate it. Build it
and HAVE FUN!