



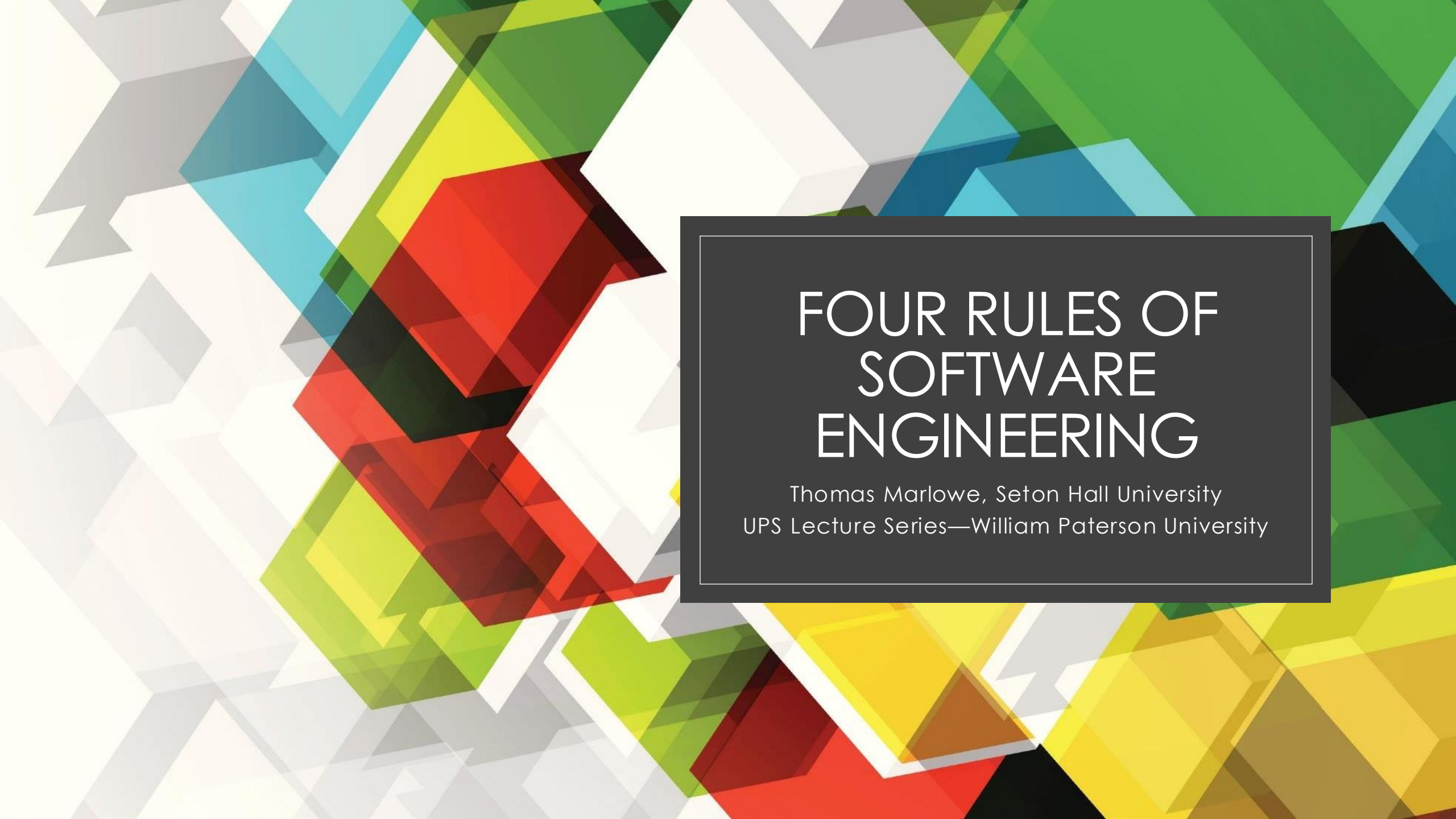
CHARLES H. FRANKE MEMORIAL LECTURE

Charles H. Franke Memorial Lecture, Seton Hall University, 2024
UPS Lecture Series, William Paterson University, 2021

UPS Lecture--William Paterson University--April 16 2021

Some reminiscences of Charlie Franke

- I've posted some memories of Charlie and his influence on me and my academic career
 - I'll just mention two
- Teaching from memory
 - Charlie never used notes
 - Only once did he go back to his office for 5 mins to consult
- Going back for CS
 - As the department expanded to include CS, Charlie went back for an MS
 - He then encouraged me, which started me on the road that led here



FOUR RULES OF SOFTWARE ENGINEERING

Thomas Marlowe, Seton Hall University
UPS Lecture Series—William Paterson University

Note

- Presentation is oriented toward computer science and software development
- Not hard to translate much of it into guidelines for the process of
 - Mathematical modeling
 - Statistical modeling and design of experiments
 - A data science/data analysis application
 - Interacting with generative AI and prompt engineering to solve a problem
- Further brief discussion at end of presentation



RULE 1

It Doesn't Start with the Coding

Providing a context

A software development problem

- Sort a set of student records
- Student name, campus ID,
major code, number of credits, GPA
 - Plus list of current courses
 - Each course has identifier, name,
instructor, number of credits
- Simple enough that we can try to do it as a single development step



A software development problem

- What do you need to know?
- Assume that there are no external constraints on
 - Programming language, IDE, tools, other resources
 - Development methodology or team structure
 - *Except as affected by the problem itself, or by testing requirements*
 - So don't consider
- Suggestions?



How did I get my list (and my lecture)?

- Not fully formed just by looking and writing
- Teaching SW engineering, data structures, databases, statistics, ..., for years
- Learning from texts, professional books & blogs, conference papers & posters
- Interaction with colleagues and with students
- Repeated reflection and modification
- At least 50 content changes in this deck from first development to this presentation
- It would probably change more if I had more time



Some questions—Input

- How many records? How big is each record? Does size vary?
 - Do we have all the records at the start?
 - Where do we get the records?
 - Local or remote file/files
 - Received individually (user input, streams, ...)
 - Can records be modified, or are they fixed?
 - Same order, encoding and meaning for fields?
 - Is input assumed legitimate (type and range consistent, etc.), or do we need to check?
 - User interface—*¿Solamente en ingles?*
- Affects choice of sorting algorithms and data structures
 - Affects design of interfaces
 - With modification may want database
 - May require preprocessing

Some questions—Processing



- ***What are we sorting on?***
 - ID #, Name, other, user selection? A tuple (major, ID)?
 - Single pass or repeated—interacts with static or dynamic data set
- Do we need to keep the original data set in original order?
- What do we do with duplicate records?
 - Same exact information, refinements (same or missing), clearly same but newer, conflicts for same key
- What do we do with invalid data?
 - Invalid records, invalid fields, constraint violations

Some questions—Output

- How should the results be presented?
 - Local/remote monitor(s), local/remote file
 - Can the user filter the results?
 - How should accessibility be handled?
- If output is presented, what report/display structure is required?
 - Will all the output be available to every viewer?
- If output is stored, what structure should be used?
- Does the output need to be preserved?
 - Very important if input is dynamic or modifiable—timestamps
- What needs to be done for accessibility?



Some questions—Security

- Security and related topics
 - Is the data subject to privacy/confidentiality constraints?
 - Should fields be suppressed/transformed in display of results?
 - Do we need extra security for storage/computation/communication?
 - If either input or output are stored, should they be encrypted?
 - Is access control/validation needed on inputs?
 - Is validation needed to view output?



Some questions—Other concerns

- Other extra-functional constraints
 - Are there criteria for quality, availability, timeliness, ...?
 - Does there need to be documentation, a user manual, ...?
 - Are there ethical or social issues? Legal issues?
- Meta-questions
 - How long is this application going to last?
 - Are we responsible for fixing problems?
 - Are we going to need to extend this?
 - Is this for internal use, for a client, or for sale?
 - How much documentation/help will we have to provide?



Not all of these
are fully germane
for this problem



RULE 1

It Doesn't Start with the Coding

The Bigger Picture



BE PREPARED

It isn't just getting the credential!

Charles H Franke Lecture—Seton Hall University--April 26 2024

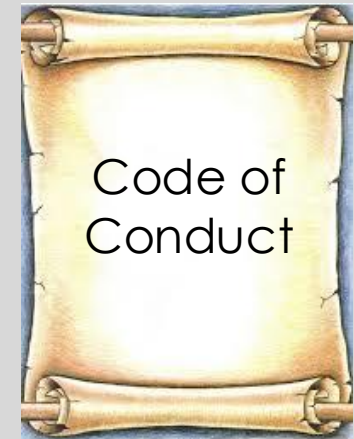
The Long View—Preparing for a Career

- Technical knowledge
 - Programming, IDEs, software engineering (traditional, Agile, DevOps)
 - Data structures, algorithms, databases, networks, operating systems, ...
 - AI and Data Science
- Professional competencies
 - Communication, teamwork, leadership, empathy, ethics, standards
 - Critical thinking, problem solving, brainstorming, learning to learn
- CS or math has prepared you very well
 - Coursework, team projects, undergraduate research, internships
 - Projects become “bigger”, and so has your responsibility within them



The Long View—Preparing for a Career

- Understand your responsibilities
 - To your employer, your customer, your fellow workers, the software development community, and society
 - And the responsibilities others have toward you
 - ACM Code of Ethics (<https://www.acm.org/code-of-ethics>)
 - ACM Code of Software Engineering Ethics
<https://ethics.acm.org/code-of-ethics/software-engineering-code/>
- Combine generalist skills with developing a specialty
 - Consider professional memberships and certifications



The Enterprise View—Working Together

- You're going to work in a context and with others
- Understand the company
 - And its management and technical processes, at least at a high level
- Understand job responsibilities (and be willing to change)
- Understand team structure and practices
- Be willing and able to cooperate for success and team building



The Work View

- OK, you've got a job as a software developer (Hooray!)
- You're working on a large project with a team
 - Perhaps communicating with other teams
- The method may be agile, traditional, hybrid, or specialized
- So, we can just start coding?
- Well ...





LOOK BEFORE YOU LEAP

Make sure there's water in the pool!

Charles H Franke Lecture—Seton Hall University--April 26 2024

The Project View—Before we start

- Should we be doing this at all?
 - Does it fit with our expertise and business goals?
 - Do we have the resources and training?
 - If for a customer,
do we feel comfortable in the relationship?
- Do we know what the problem is doing—large scale?
 - Product Vision
 - What problem is it trying to solve, and how will it try to solve it?
 - Business Case
 - How does the solution fit with our business? With the customer's?



The Project View—Before we start

- Do we have the right resources?

- **Time and money**

- Personnel

- Training and support

- Experts as needed

- Tools—for development, for management, for communication

- Do we have the right team? Do we have the right process?

- Do we have management backing on project, process, and product?

Tradeoffs in when we investigate,
when we stipulate,
and when we acquire resources
Don't expect full answers right away



The Project View—Before we start

- What risks might we face?
 - To project, process, product, personnel
 - External risks and catastrophes
- Do we have an RMMM plan and structure?
 - Monitor, manage, and mitigate risk
 - Management task with technical input
 - Avoid or prevent if possible
 - Mitigation—recovery, reduction, restoring morale and productivity, ...



More tradeoffs—but
Must anticipate major risks
Must set up structure

The Project View—Artifacts and Views

- Each workflow produces both informal and formal artifacts
 - Code is one—and incorporates others like comments
 - Others come from tools or analyses—like unit test reports
- Artifacts may need views
 - Role Team, management, customer, users, operators/helpdesk
 - Level Precise, overview, details
 - Aspect Behavior, interfaces, security & privacy, timeliness, ...
- Understand what belongs/will be found where





AND GET TO WORK

You may specialize, but you need a global view

Requirements—Understand the problem

- What do we need to know at the start (of this pass)?
- What definitely needs to be considered?
 - What must be done? What are its interactions?
 - What's going to be hard? Tricky? Uncertain?
 - Subject to frequent change?
 - How does it need to do it? What about
 - **Security, privacy/confidentiality, intellectual property?**
 - **Timeliness and safety** (active systems), availability, resilience, performance?
 - Business & technical services—accounting & inventory, logging & access?
 - Support—manuals, documentation, on-line, **accessibility**, ...?
 - What legal, standards, and ethical issues may arise?



Requirements—Understand the problem

- How does this add value for the customer? The user?

- For example

- Behavior

Increased functionality

Better performance, reliability/resiliency, ...

- Interaction

Improve or add interfaces & improve accessibility

Better understandability, usability, aesthetics

Reduce annoyance/inconvenience

- Extensibility

Support additional platforms

Improve internationalization, interoperability

Address product line consistency

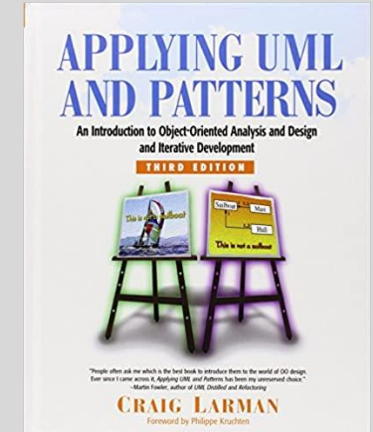
Requirements—Understand the problem

- How do we learn about requirements?
 - Discuss and listen
 - Read (regulations, standards, terminology, common practice, ...)
 - Observe
- Check interpretation vs problem & w/ customer (!)
 - And end users/operators if possible
 - Test if possible
- What about problems and constraints?
 - What is missing? What works poorly?
 - What has gone wrong? How and why?
 - Are we introducing new concerns?
 - What needs to be guaranteed?



Specification—Understand the workings

- How do the business processes work?
 - Happy path(s)—when everything goes right
 - Alternative flows—variants and options
 - Handling special cases, exceptions, errors — existing and new
- Our product may provide same services very differently
- What needs special attention?
 - Try to capture implicit assumptions and tacit knowledge
 - Understand concurrency/non-determinism in problem
 - What can happen in parallel? In different order?
 - Where is security an issue? Where are the bottlenecks? Precise timing?



Emphasize
when unclear, tricky,
or complicated

Modeling—Understand the structure

- Isolate real-world entities and interactions
 - Some entities are not physical — transactions, meetings, ...
- Create conceptual objects and information flows to handle
 - Effort focused on less-than-obvious translations &c
- Include (at least) security, privacy, timing, safety, accessibility from start
 - Need not be fully implemented
 - But need to be aware and leave flags and hooks
- Be aware of points of fragility—Where can this break?



Design—Understand the solution

- The application world is a model of the model
 - Make a further translation to that model
- Create skeleton for method code
 - Decide on algorithms and data structures
 - Use design patterns, guidelines, and idioms
 - May modify modeling/domain level, largely in predictable ways
 - Example: materializing composite objects/collections
 - Add hooks for essential services—but don't go overboard

In many cases, this isn't
much different from
creating a code skeleton

So

- We've done all that!
- All that's left is the coding, right?





RULE 2

It Doesn't End with the Coding

In the real world, you can't just turn in your assignment

OK, what's left?

- You're a genius coder
- You solved the problem with your team
- Everyone should be smiling
- What have we left out?

What do we need to do now?

- Any ideas?



Using Generative AI

- It has become increasingly common to use generative AI
 - To generate code for common functionality
 - To serve as a pair programmer
 - To generate documentation and help
- Responsibilities of the user
 - Don't use or use with extreme care for critical code
 - Review & test the artifacts—check for correctness and completeness
 - Beware of hallucination/fabulation



Testing—Check the solution

- Goals

- Do the right thing

Behavior corresponds to requirements
Constraints are met



- Do the thing right

Implementation is effective and efficient
We aren't going to get in trouble for it



- Do the thing well

Standards are met
Application can evolve



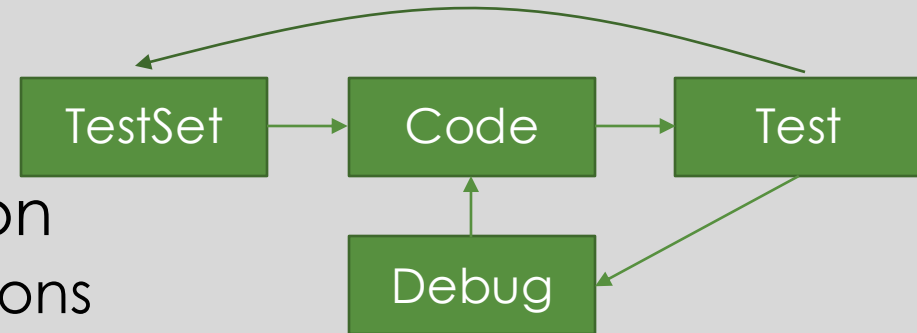
- Keep doing it better

Test/validate the process as well



Testing—Check the solution

- Test-driven development
- Integrate testing with implementation
 - Anticipate with prototypes or simulations where appropriate
- Make sure to test for error paths and systems errors
- Some testing occurs late
 - Stress testing—overload the application or the system
 - Platform testing—test in actual deployment environment
 - Acceptance testing—run working version past customer/users



Validation & Verification—It ain't just data

- Not just testing tools and checking interactions
 - Code reviews, static/dynamic analyzers, metrics
 - Security testing, timeliness, privacy protection, ...
- Validation
 - Compare to specification and requirements
 - Ensure that safety/health and legal standards met
- Verification
 - Address mathematical properties
 - Protocols, algorithms, ...
 - Ensure proper sequencing of operations and proper values



Maintenance—Assure the solution

- Address user issues and other problems that arise
- Sometimes just a matter of communication
 - Point in the right direction
 - Clear up misconceptions
- May require modifying models, code, or other artifacts
 - Misunderstandings may suggest changes
 - Changes might be to documentation or manual rather than code
 - Some changes can be deferred to next update/release



Maintenance—Assure the solution

- What changes might we have to make?
 - Corrective—debugging and reaction to test issues
 - Adaptive
 - Respond to platform/system changes
 - Support interoperability and portability
 - Preventative
 - Changes needed for security, privacy, and related problems
 - Perfective—make it better
 - Appearance, understanding/learning/use, performance, ...



Maintenance—Assure the solution

- What changes need to be immediate?
 - Corrective—Significant errors in behavior that
 - Break the application
 - Endanger life, health, environment, ...
 - Require only minor, transparent changes, or changes to documentation
 - Preventative—Security holes, privacy violations, legal issues
 - That have occurred, are public knowledge, or are likely to occur



Evolution—Extend the solution

- A major focus of agile methods and DevOps
 - Development is sequence of iterations & releases
- Embraces major adaptive and perfective changes
- Move
 - Into new versions and variants of the product
 - From this product to related products or product line
- Adjust to changes in user community or new uses



Reflection—Improve the process

- What went right? What went wrong?
- What felt uncomfortable?
 Could have been simpler?
- Aim to improve
 - The development process
 - Development and management practices and culture
 - Uses of artifacts and tools
 - Team culture and interactions
- Reflect on social, ethical, ... issues



So

- OK, we're ready for that!
- Now, all that's left is the coding, right?





RULE 3

There's More than Coding
While You're Coding

Preserving Sanity

Charles H Franke Lecture—Seton Hall University--April 26 2024

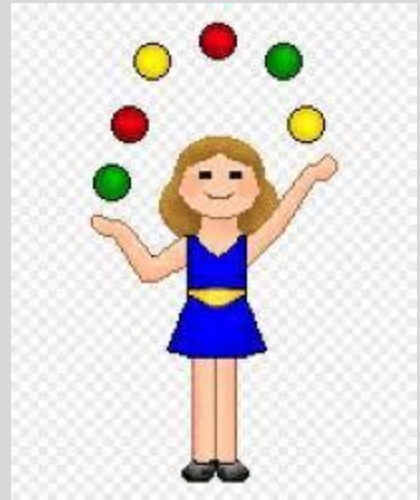
You're not alone in the world

- Keep your team aware
 - Of what you've done, are doing, and any choices you have made
- Often structured and supported
 - Institutional and team practices
 - Team meetings such as Scrums—address all workflows
 - Pair programming in XP
 - One writes code, other checks/critiques/comments/documents
 - Recent observations—don't let it become a bottleneck



Keep all the balls in the air

- In modern approaches to software engineering
- Coding is interleaved with other workflows
 - Problems and opportunities discovered in coding
 - Incremental incorporation of features and support
- In agile and related methods
 - Interleaved with refinement of requirements and specification
 - Across multiple incremental passes
 - Addressed for example in the (Rational) Unified Process, based on UML artifacts and practices



Repeating myself

- Follow institutional and team testing practices
- Write test code first (or at least understand how to test)
 - As a team, results in development of a test suite for the application
- Sometimes need to use *mocks* or simulations
 - For components that have not yet been developed
 - For external components
 - If actual execution is dangerous
(cardiac workstation, atomic plant), impossible (Mars Rover), hard to repeat
- As usual, non-determinism/concurrency and timeliness require extra care



This isn't just an assignment

- This is an ongoing, dynamic project
- A large project might have
 - Millions of lines, thousands of methods, hundreds of classes, ...
 - Dozens of consistent versions, and of families of variants
 - A lifetime of ten years or more
- Code may be under revision by multiple team members
- To keep track, need to interact with
 - Version control
 - Configuration management and history
- Keep documentation and manual consistent with code



After six months, you too will a
stranger to your own code
and documentation



RULE 4

Don't Forget About the Coding

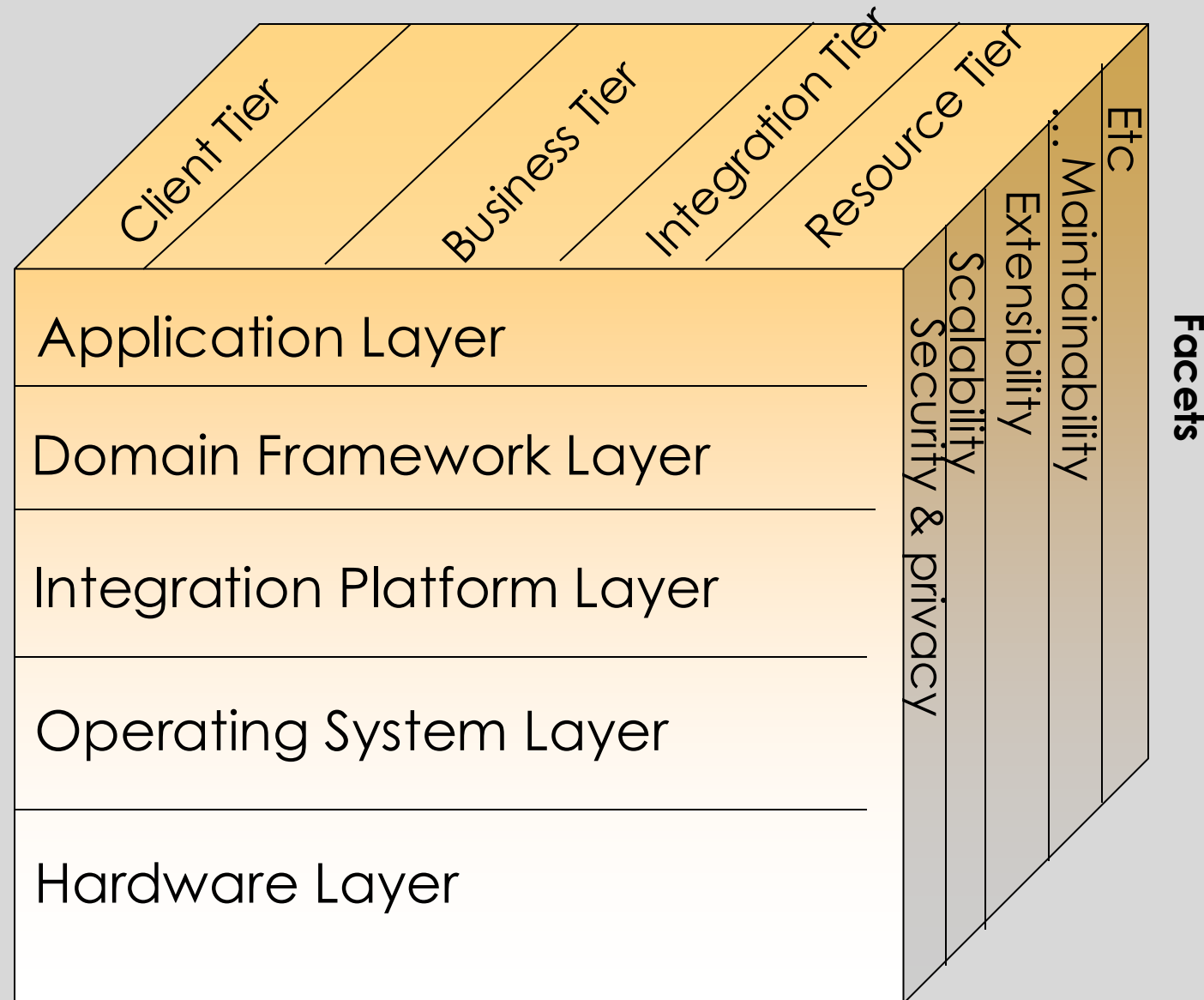
Good Practice is what makes Perfect (or nearly so)



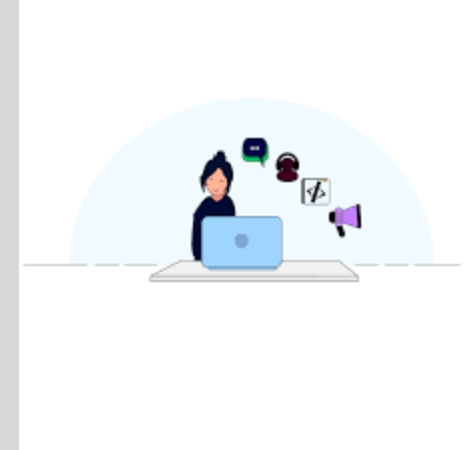
Layered development



- Layer, tier, facet
 - Layer How close to “the surface” ? UI, business logic, services
 - Tier What part of the solution? presentation, rules, ...
 - Facet What issue is addressed? security, availability, ...
- May be different tools for layers, services/aspects for tiers/facets
- Underlying infrastructure layer
 - Databases and operating system services
 - Azure/AWS/gcp — PaaS Web services
 - Thick client install, Web-based SaaS, microservices, the cloud



Full Stack development



- Modern model of web development

- Full stack

- Front end/client

User interface, communication, environment

- Back end/server

Business logic

- Database

Data, transaction management

- Services

Includes OS, network and other technical

- Two examples—order to make acronym neat, not layering

- MEAN Stack

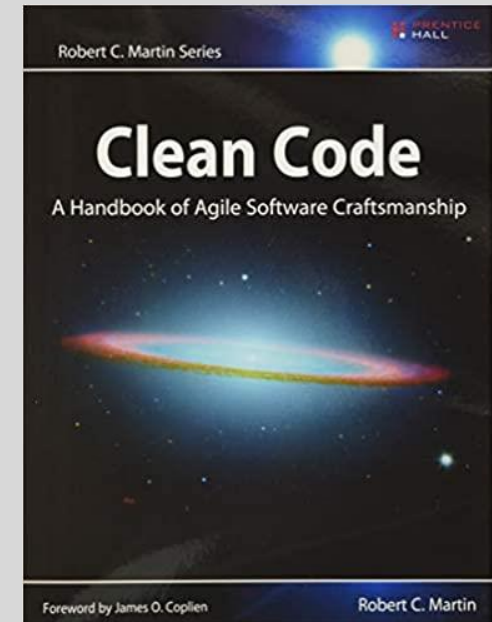
MongoDB, Express, AngularJS, Node.js

- LAMP Stack

Linux, Apache, MySQL, PHP

Coding Standards

- Write good code
 - See Robert Martin's **Clean Code** as a resource
 - Great book!
 - Simple modules, classes, and methods
 - Well-structured, simple interaction
 - Good, informative names
- As before
 - Follow team and enterprise guidelines
 - Aim at good comments and consistent documentation
 - Pay attention to concurrency and non-determinism



Coding Standards

- Don't screw up the future
 - Cute tricks often screw up optimization and inhibit modification
- But don't overdesign for the future
 - Much of the code will never be changed in major ways
 - And many changes will follow well-known patterns if needed



But!

- Anticipate interfaces/services may vary
- Use design patterns to minimize or localize dependence on
 - Different user interfaces
 - Different services for different variants, situations, ...
 - Different implementations of services and collaborators
 - Different algorithms or data structures to implement behavior
 - Local vs remote connections or data
- Adapter, Façade, Proxy, Decorator, Observer, Command, COR



Danger Signs

- Be very careful changing code that may interact with
 - Security, privacy, and access control
 - Timing constraints, and safety and health
 - Non-deterministic behavior and concurrency
 - Performance and resiliency with major effects
- Use code reviews, traceability tools, analyzers, and testing to see what is affected by error or change, and where to make changes
- Regression testing is a key tool and should be automated
 - Provides assurance that changes have not broken behavior guarantees



Speed can kill



- Ordinarily, we want to keep up the pace
 - Rapid development key to modern methods
- But if you're working on something tricky, risky, intricate
 - Talk about it first!
 - Test-first-development essential here
 - Flesh out a skeleton
 - Loop: Discuss, reflect, develop, test, debug
 - Sometimes reflection means you actually “need to sleep on it”

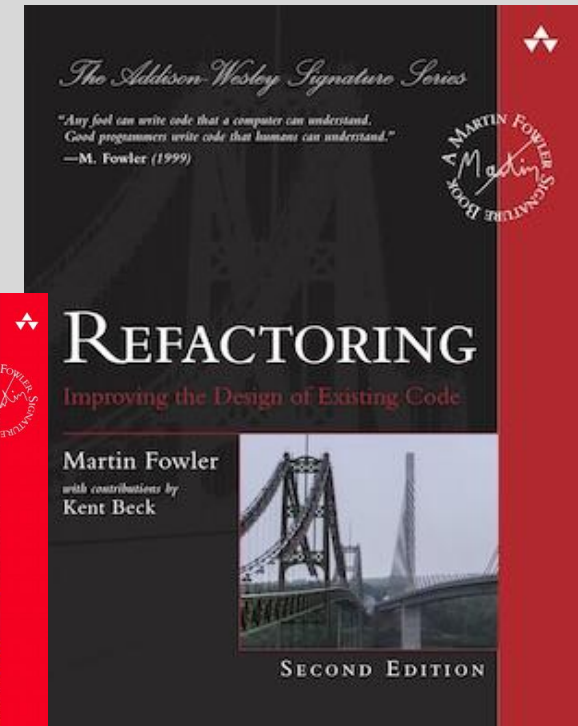
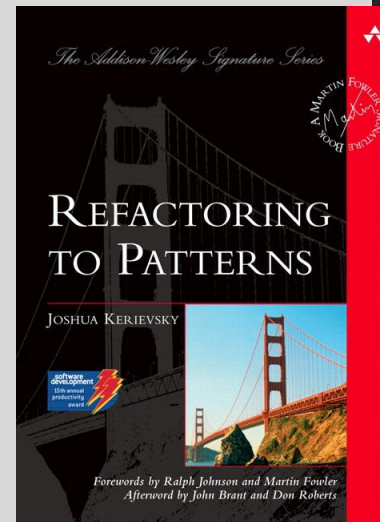
Technical Debt and Refactoring

- Technical debt
 - Code tends to become ugly as it is changed
 - Some code is no longer needed or no longer used
 - Some code works but is confusing
 - Comments and documentation become inconsistent with code
 - Structure of code does not reflect its current function
- Further changes more likely to introduce bugs or confusion



Technical Debt and Refactoring

- Address by local refactoring if possible, redesign if necessary
 - Refactoring uses *code smells* and bugs to identify issues
 - And design patterns to guide sequences of local transformations
- A sequence of refactorings can fix most problems
- A large-scale rewrite is a last choice



Don't bother if it's not worth it

- Focus attention on
 - Code implementing core functionality—as discussed before
 - Meaning of “core” changes as product matures
 - Code that frequently breaks, needs rewriting, or has issues
 - Code that will need to be generalized
 - Identified threats
 - Interactions with the outside



While you're coding

- Add (only) meaningful comments
- Document your changes
(at an appropriate granularity)
- Write document comments (Java ///)
- Make note of
 - Open issues
 - Need for documentation
 - Ugliness in the code—includes inconsistencies with practices

```
/**  
 * Code Readability  
 */  
if (readable()) {  
    be_happy();  
} else {  
    refactor();  
}
```

Evolution revisited—Extend the solution

- We're never done until the product is retired
- Completing the product we envisioned
 - Go back to slide 28 😊
- Improving the product we envisioned
- Designing variants and versions
- Fitting better into a product line or larger system



Evolution revisited—Extend the solution

- Incremental and iterative development

- Iterative delivery
- Versions and variants
- Product line
- Configurability
- Accessibility

- Tree of products

- Many early products still sellable

Iterative Development of a Chair

- Version 1 Wood base, 4 legs
 - Variant Child seat
 - Product line Bench
- Version 2 Add back and fabric cover
 - Configure Carpentry style
 - Variant Stool
- Version 3 Add arms
- Version 4 Upholstered base, arms, back
 - Configure Pattern, stain resist
- Version 5 Pillows, optional monogram, ...



META-RULE 1

No silver bullets, no golden hammers

Procrustes' Bed is not a comfortable place

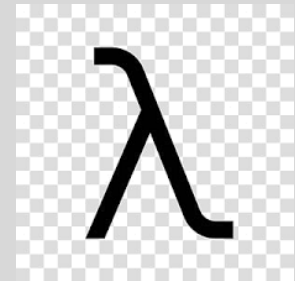
Fit your method to the problem

- Writing a linear-algebra algorithm tool is not the same as writing an intelligent recommender system
 - Requirements are not likely to be dynamic
 - Outside of user and system interfaces
 - Constraints typically on performance or applicability
 - Code chunks for well-understood computations can be larger
 - They won't change (if correct) and they won't be misunderstood
 - Parameter lists can be longer
 - Although it may still make sense to pack them in an object
 - It may make sense to call variables x, y, z, and t



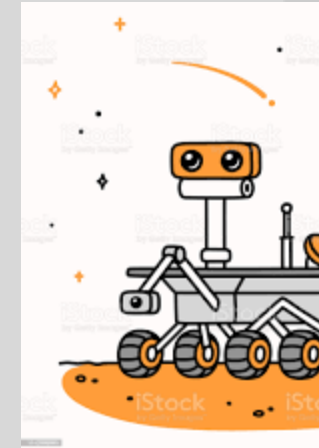
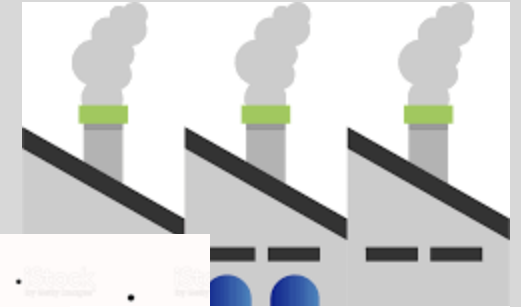
Fit your method to the problem

- Some pieces may call out for a different coding paradigm
 - Functional code
 - Use of aspects to implement cross-cutting concerns
 - Basis of Kotlin and Scala, and now partially supported in Java
 - Logic code and pattern matching
 - Interactions with a database, even when data isn't originally in one
- Active and real-time systems always require method and tool enhancements / extensions
- Current + future discussions on the role of AI in code generation



Fit your method to the problem

- Lots more to say—some mentioned before
 - Physical constructs like chemical plants
 - Cardiac workstations and atomic power plants
 - The Mars rovers
 - Even financial tools and healthcare records
- Each has different constraints on
 - What can be modified and when
 - What can be tested in the real world—especially extreme situations
 - Whether iterative solutions can be deployed—and where they apply





META-RULE 2

It's an approach, dammit,
not a religion!

We're training critical thinkers, not Bible salesmen!

Don't complicate without reason

- Support future change only where change is likely
- Don't use indirection where only one instantiation is likely
- And so on



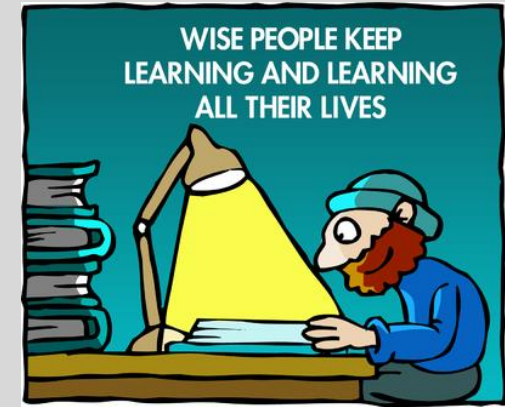
Don't be afraid to break the rules

- If it doesn't matter
and it makes things much simpler
- If it does matter but
following the rules would create a mess
- But don't do that without
 - Informing your team, and
 - Leaving a comment—that you've done it and why
- Understand how you could redesign or refactor if necessary



Keep learning

- Don't become too attached to how you do things now
- While processes and practices improve *on the whole* over time
 - That doesn't mean it's perfect now
 - Some things might be worse than they used to be—it's a transition
 - And others will be replaced by what's better or more effective
- Changes in technology and in theory (!) will affect what you can do and how you can do it
 - And you want to keep up





META-META-RULE

It's Not Just
Software Engineering

It's here, it's there, it's everywhere!

These Rules Carry Over

- With appropriate changes in wording and concepts
 - Fancy Latin *mutatis mutandis*—changing what needs to be changed
- Also apply to
 - Mathematical modeling of complex, changing problems
 - Problem solving of ongoing problems
 - Preparing a presentation
 - Investment portfolio analysis and management
 - Project management and agile business processes
 - Security including cybersecurity
 - Much critical thinking (and learning critical thinking)
 - And more



Some important references

- Build yourself a library—print, virtual, or a mix
- Robert Martin, **Clean Code**
 - All of “Uncle Bob’s” stuff is exceptional
- Martin Fowler, **Refactoring**, <https://martinfowler.com/>
- Johnson, Helm, Gamma, Vlissides, **Design Patterns**
 - Lots of follow-up books, articles, web pages, ...
- Craig Larman, **Applying UML and Patterns**
- Jeremy Kubica, **Best Practices of Spell Design** (fun view of clean/agile)



Acknowledgments

- Dr. John Saccoman and Prof. Tara Wager
and the Department of Math & CS

- Research colleagues

- Fr. Joseph Laracy
- Cyril Ku
- Vassilka Kirova
- Garrett Chang
- Om Hashmi
- Stephen Masticola

SHU Math & CS + School of Theology

William Paterson University

Bell Labs Consulting (retired) & NJIT

President, Highstep Technologies

Vice-President, Agile Brains Consulting

Siemens Technology (retired)



and the
Tech staff!

Questions?



APPENDIX: THE BROAD VIEW AND THE STANDARD CS CURRICULUM

Charles H Franke Lecture—Seton Hall University--April 26 2024

Workflows—A lot of detail

Software Project	
Business Model—Business case, Product Vision	Umbrella Activities & Management Resources, Training, Risk, ...
Requirements—Elicitation, acquisition, analysis	
Specification—RW patterns of interaction	
Modeling—Use cases and Domains	
Design—Objects, communication, patterns	
Implement—Data structs, algorithms, details	
Test—Unit, integration, system, stress, platform	
Maintenance—Correct, adapt, prevent, perfect	
Evolution—Extend, refactor	
Reflection—Lessons learned, needed change	

Workflows—Evolution during your major

Introduction		Software Project	
		Business Model	Umbrella Activities & Management
		Requirements	
		Specification	
		Modeling	
		Design	
Implementation		Implementation	
		Testing	
		Maintenance	
		Evolution	
		Reflection	

Workflows—Evolution during your major

Data Structures			Software Project	
			Business Model	Umbrella Activities & Management
			Requirements	
			Specification	
			Modeling	
	Design		Design	
Implementation			Implementation	
Testing			Testing	
			Maintenance	
			Evolution	
			Reflection	

Workflows—Evolution during your major

Databases			Software Project	
			Business Model	Umbrella Activities & Management
	Req'ts		Requirements	
	Spec		Specification	
	Modeling		Modeling	
	Design		Design	
Implementation			Implementation	
Testing			Testing	
			Maintenance	
			Evolution	
			Reflection	

Workflows—Evolution during your major

Software Engineering			Software Project	
	Maybe		Business Model	Umbrella Activities & Management
	Req'ts		Requirements	
	Spec		Specification	
	Modeling		Modeling	
Design			Design	
Implementation			Implementation	
Testing			Testing	
Mainten			Maintenance	
Maybe			Evolution	
			Reflection	