Bune practici & Refactorizare

Mihai Angheluță
Technical Lead @ SPGlobal

S&P Global

Happy Dance like a 🐔 day!

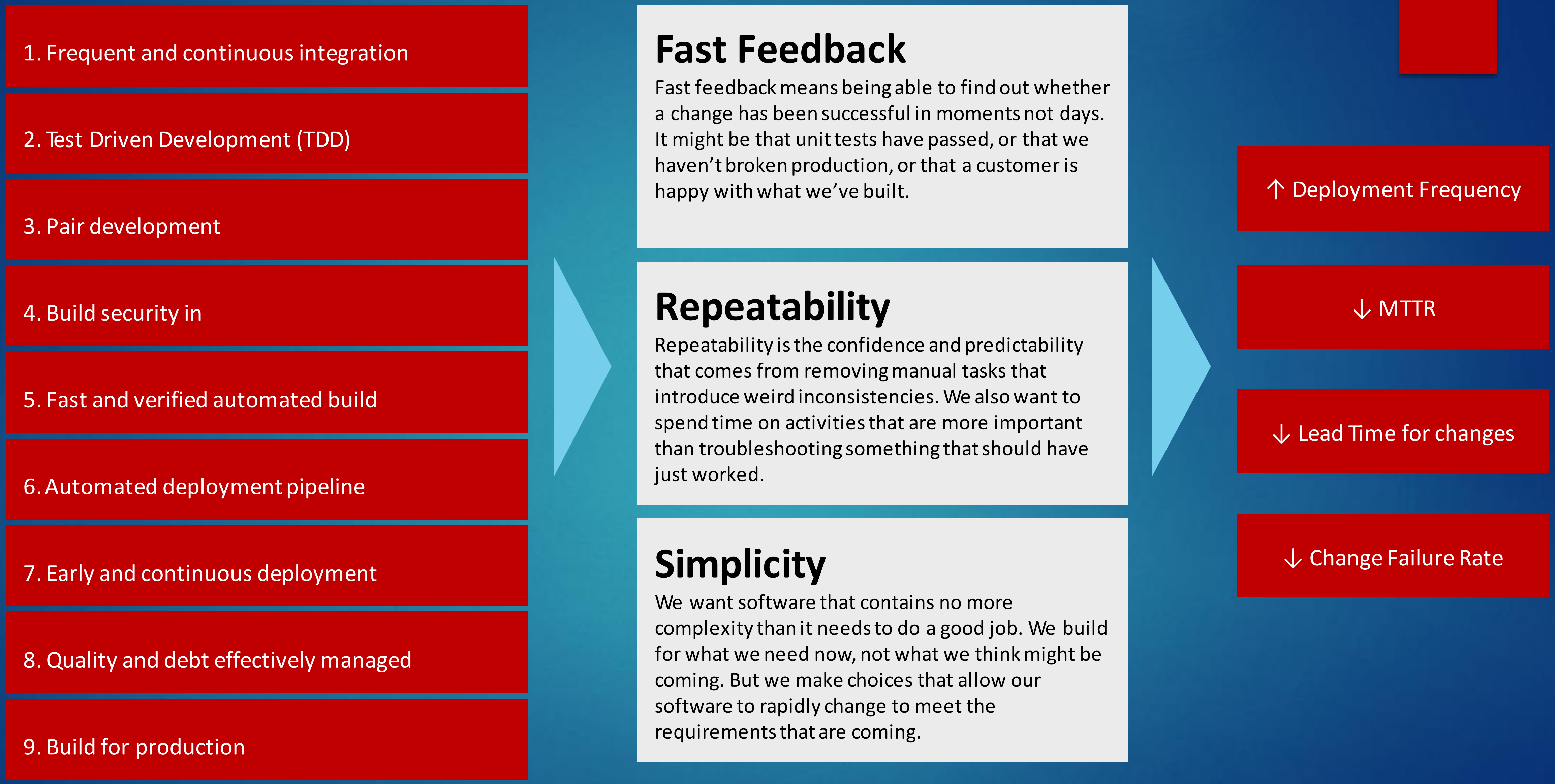
Bune practici implicite

- Frequent and continuous integration
- Test-driven development (TDD)
- Pair development
- Build security in
- Fast and verified automated build
- Automated deployment pipeline
- Early and continuous deployment
- Quality and debt effectively managed
- Build for production

Fast Feedback

Repeatability

Simplicity



1. Frequent and continuous integration
2. Test Driven Development (TDD)
3. Pair development
4. Build security in
5. Fast and verified automated build
6. Automated deployment pipeline
7. Early and continuous deployment
8. Quality and debt effectively managed
9. Build for production

Fast Feedback

Fast feedback means being able to find out whether a change has been successful in moments not days. It might be that unit tests have passed, or that we haven't broken production, or that a customer is happy with what we've built.

Repeatability

Repeatability is the confidence and predictability that comes from removing manual tasks that introduce weird inconsistencies. We also want to spend time on activities that are more important than troubleshooting something that should have just worked.

Simplicity

We want software that contains no more complexity than it needs to do a good job. We build for what we need now, not what we think might be coming. But we make choices that allow our software to rapidly change to meet the requirements that are coming.

↑ Deployment Frequency

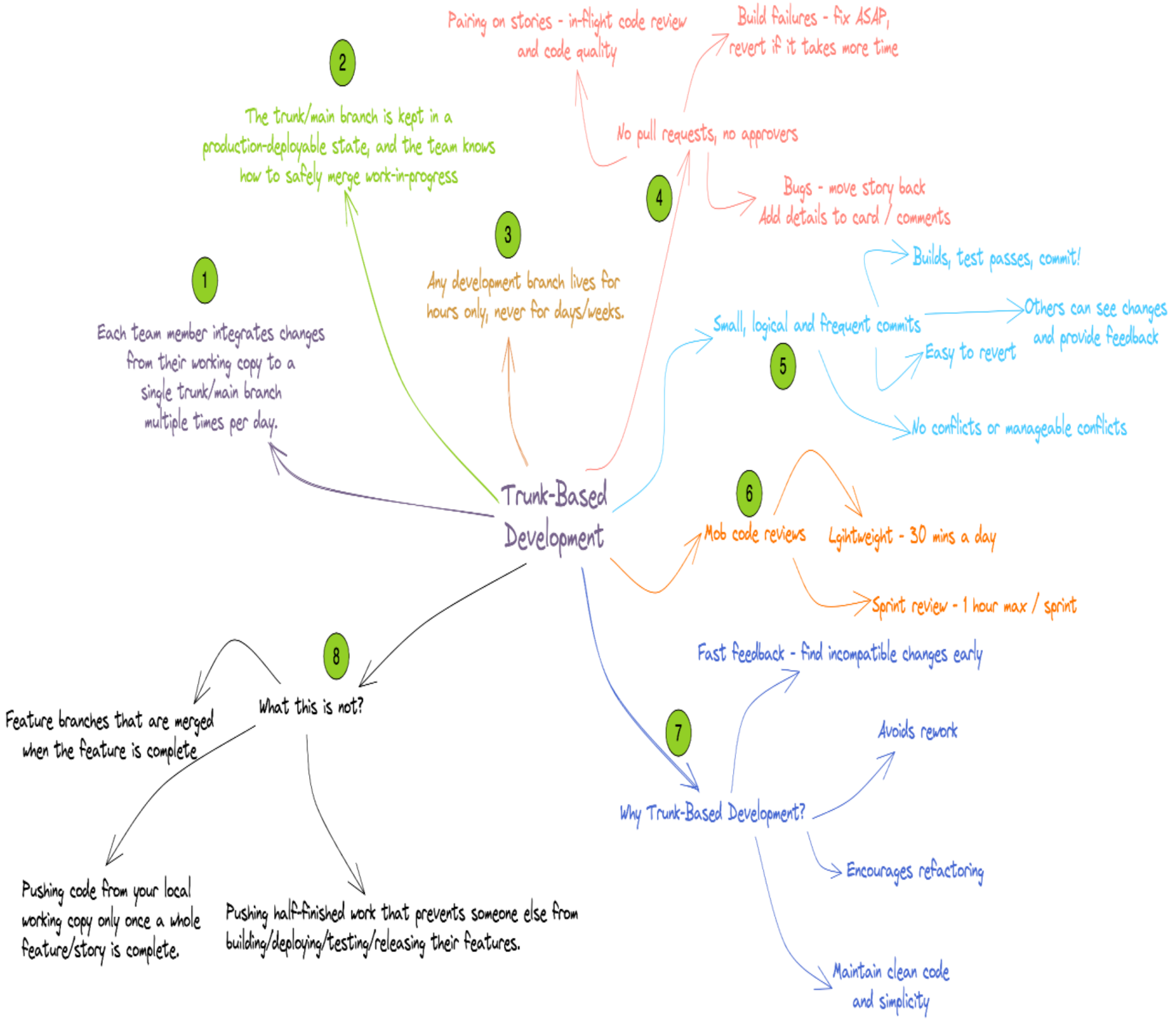
↓ MTTR

↓ Lead Time for changes

↓ Change Failure Rate

S&P Global

Frequent and continuous integration



Trunk-Based Development

1

Each team member integrates changes from their working copy to a single trunk/main branch multiple times per day.

2

The trunk/main branch is kept in a production-deployable state, and the team knows how to safely merge work-in-progress

3

Any development branch lives for hours only, never for days/weeks.

4

No pull requests, no approvers

Pairing on stories - in-flight code review and code quality

Build failures - fix ASAP, revert if it takes more time

Bugs - move story back
Add details to card / comments

5

Small, logical and frequent commits

Builds, test passes, commit!

Others can see changes and provide feedback

Easy to revert

No conflicts or manageable conflicts

6

Mob code reviews

Lightweight - 30 mins a day

Sprint review - 1 hour max / sprint

7

Why Trunk-Based Development?

Fast feedback - Find incompatible changes early

Avoids rework

Encourages refactoring

Maintain clean code and simplicity

8

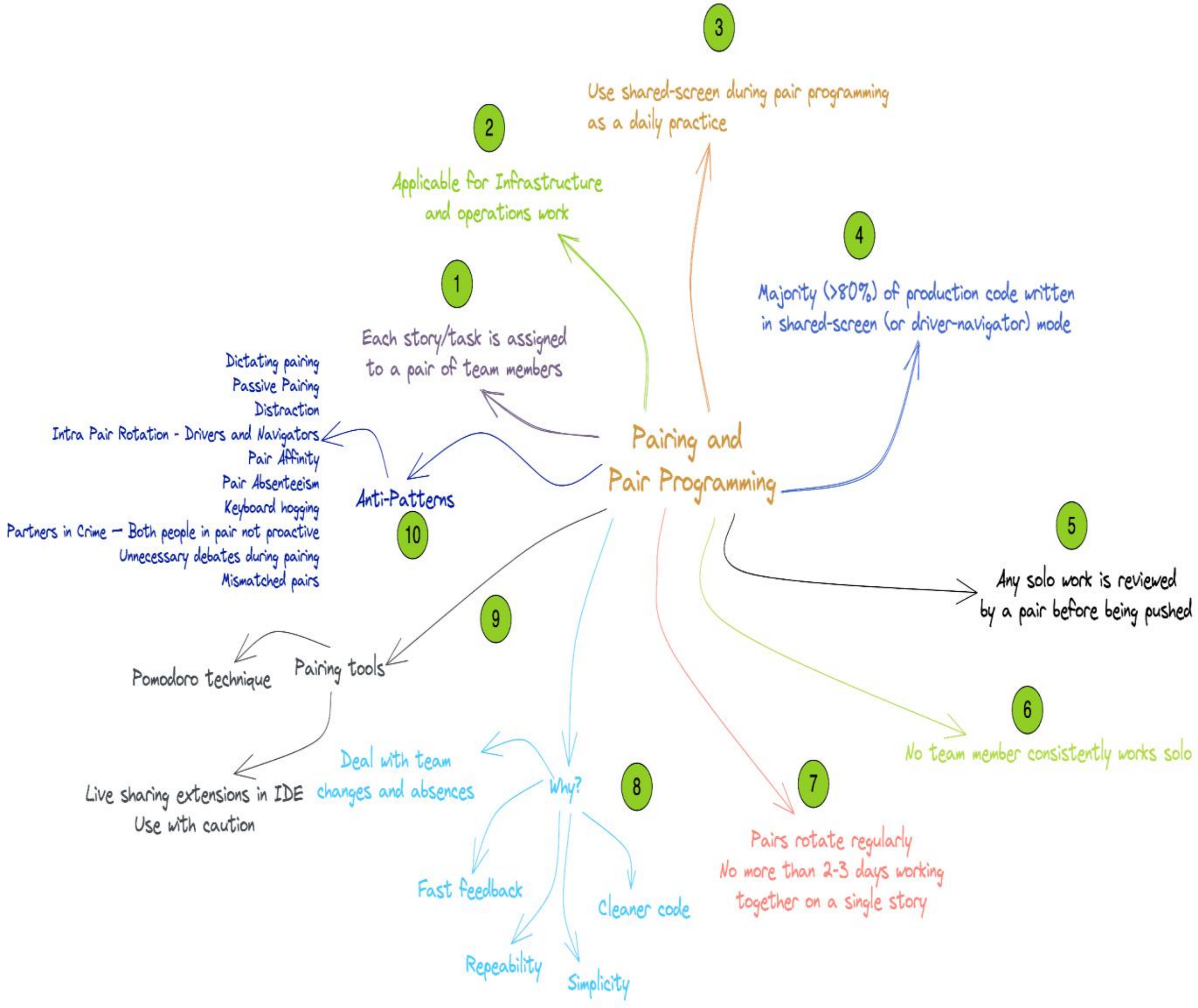
What this is not?

Feature branches that are merged when the feature is complete

Pushing code from your local working copy only once a whole feature/story is complete.

Pushing half-finished work that prevents someone else from building/deploying/testing/releasing their features.

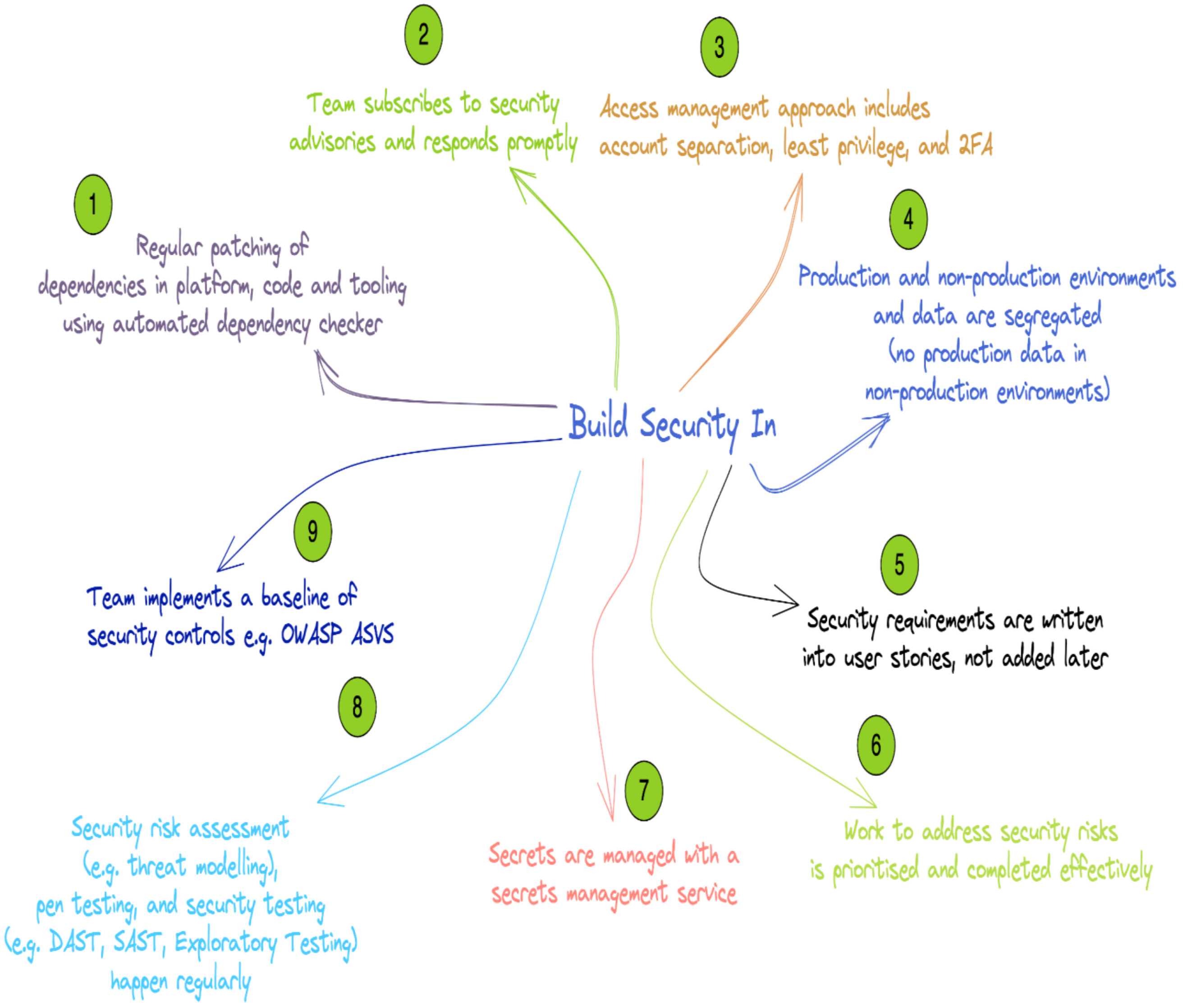
Pairing and pair programming



Pairing and Pair Programming

1. Each story/task is assigned to a pair of team members
2. Applicable for Infrastructure and operations work
3. Use shared-screen during pair programming as a daily practice
4. Majority (>80%) of production code written in shared-screen (or driver-navigator) mode
5. Any solo work is reviewed by a pair before being pushed
6. No team member consistently works solo
7. Pairs rotate regularly No more than 2-3 days working together on a single story
8. Why?
   - Deal with team changes and absences
   - Fast feedback
   - Repeability
   - Simplicity
   - Cleaner code
9. Pairing tools
   - Pomodoro technique
   - Live sharing extensions in IDE Use with caution
10. Anti-Patterns
    - Dictating pairing
    - Passive Pairing
    - Distraction
    - Intra Pair Rotation - Drivers and Navigators
    - Pair Affinity
    - Pair Absenteeism
    - Keyboard hogging
    - Partners in Crime — Both people in pair not proactive
    - Unnecessary debates during pairing
    - Mismatched pairs

Build Security In



Build Security In

1 Regular patching of dependencies in platform, code and tooling using automated dependency checker

2 Team subscribes to security advisories and responds promptly

3 Access management approach includes account separation, least privilege, and 2FA

4 Production and non-production environments and data are segregated (no production data in non-production environments)

5 Security requirements are written into user stories, not added later

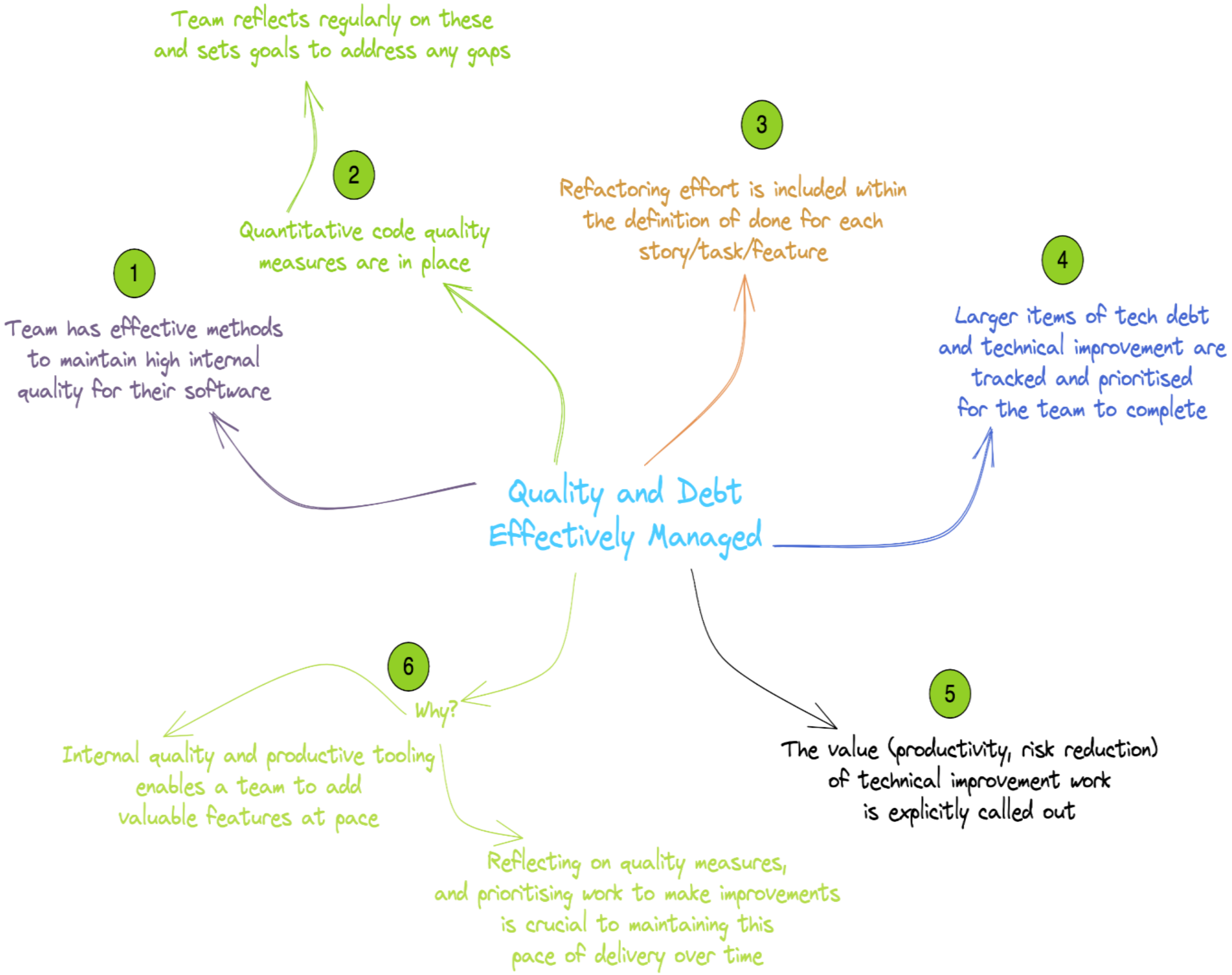
6 Work to address security risks is prioritised and completed effectively

7 Secrets are managed with a secrets management service

8 Security risk assessment (e.g. threat modelling), pen testing, and security testing (e.g. DAST, SAST, Exploratory Testing) happen regularly

9 Team implements a baseline of security controls e.g. OWASP ASVS

Quality and debt effectively managed



Quality and Debt Effectively Managed

1 Team has effective methods to maintain high internal quality for their software

2 Quantitative code quality measures are in place

Team reflects regularly on these and sets goals to address any gaps

3 Refactoring effort is included within the definition of done for each story/task/feature

4 Larger items of tech debt and technical improvement are tracked and prioritised for the team to complete

5 The value (productivity, risk reduction) of technical improvement work is explicitly called out

6 Why?

Internal quality and productive tooling enables a team to add valuable features at pace

Reflecting on quality measures, and prioritising work to make improvements is crucial to maintaining this pace of delivery over time

Ce este refactorizarea?

De ce este necesară?

Modificarea unei secvențe de cod care funcționează corect fără a-i schimba funcționalitatea externă

Parte a activităților zilnice

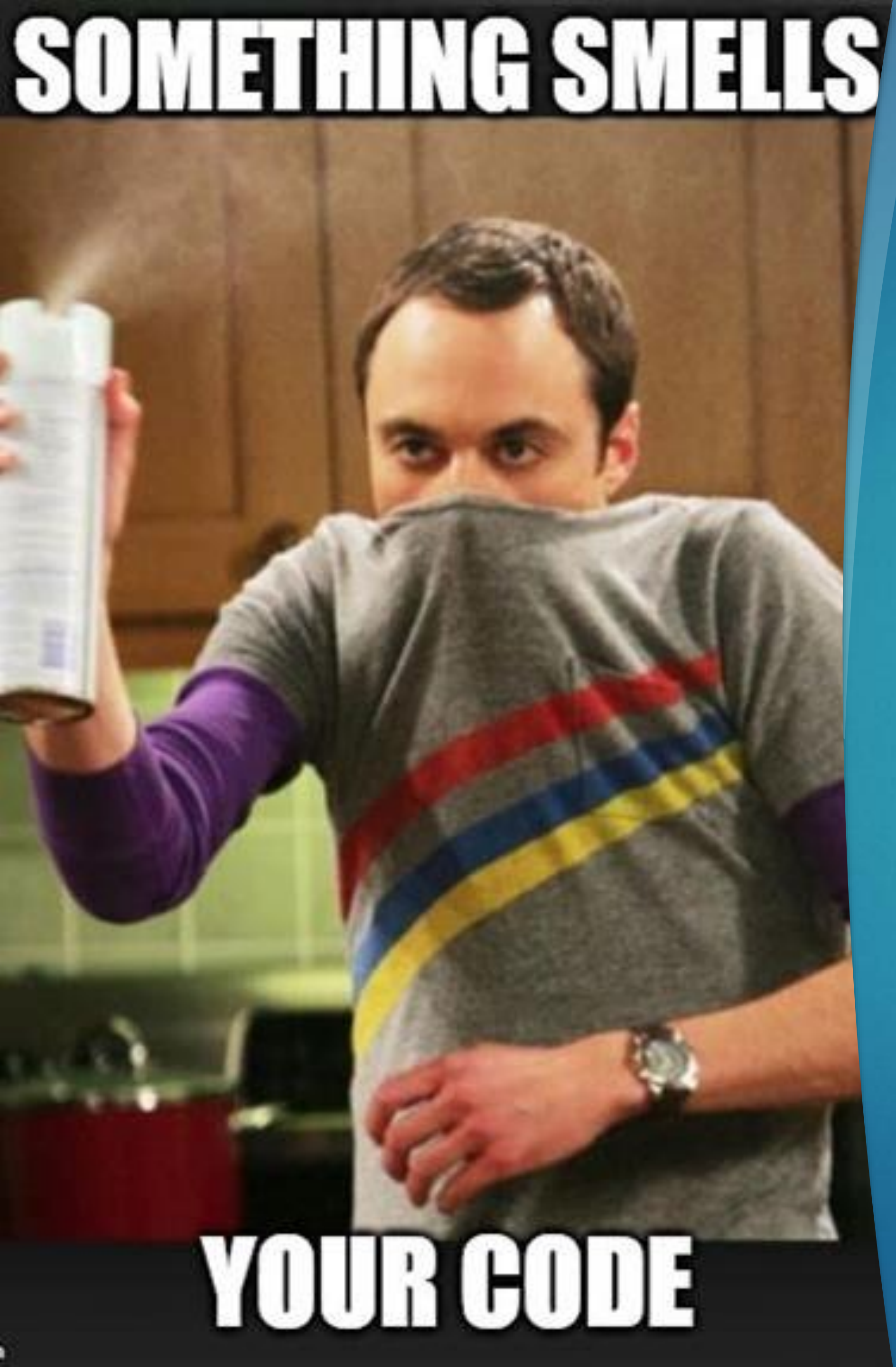
Uneltele care automatizează sunt utile dar nu esențiale

Reorganizare

Optimizare

Comportamentul extern nu e alterat

Structura internă e îmbunătățită



Code smells

La nivel de aplicație

- Duplicated code
- Shotgun surgery
- Contrived complexity
- Speculative generality
- Mysterious name

La nivel de clase

- God object
- Feature envy
- Lazy class
- Inappropriate intimacy
- Refused bequest
- Cyclomatic complexity
- Data clump
- Message chain
- Middleman

La nivel de metode

Prea mulți parametri

Metodă prea lungă

Switch sau multiple blocuri de decizie

Prea multe comentarii

Nume prea lungi sau prea scurte

Mulțumesc

Mihai Angheluță

m.angheluta@spglobal.com

S&P Global