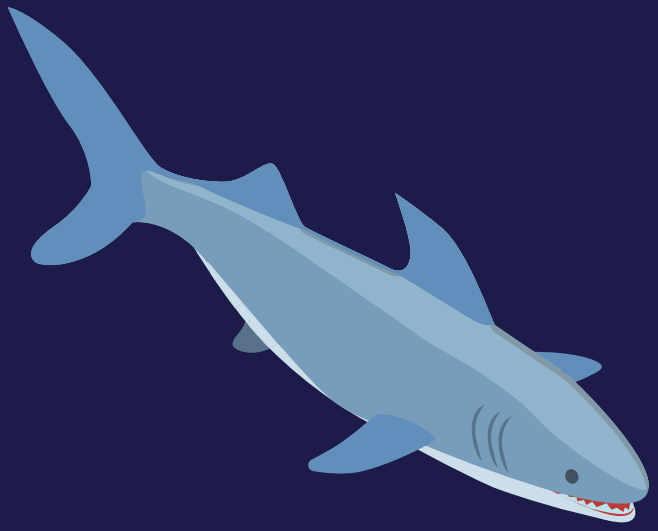
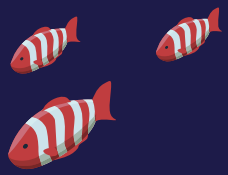
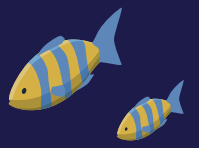
Serious Scrum RZM

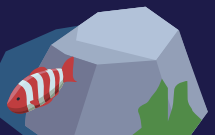
Deepdiving Developers







Road to Mastery
Virtual Edition
for Guides




Mountaineering Scrum Together
Vision
Value
Kayaking the Value Stream
Surviving Self-Management
Exploring Artifacts
Deep Diving Developer Culture
Volcanic Activity
The Valley of Values
The Game of Scrum
Agile Backpacking
Smooth Sailing the Events
Bootcamp
Coaching Dojo
Basecamp
ROAD TO MASTERY

Deepdiving Developer Culture

RIG FOR DIVE

Deepdiving Developers

Scrum is often mistaken for an engineering process. But Scrum also encompasses discovery and delivery. It's about developing ideas and creative problem-solving as much as it is about creating increments of value.

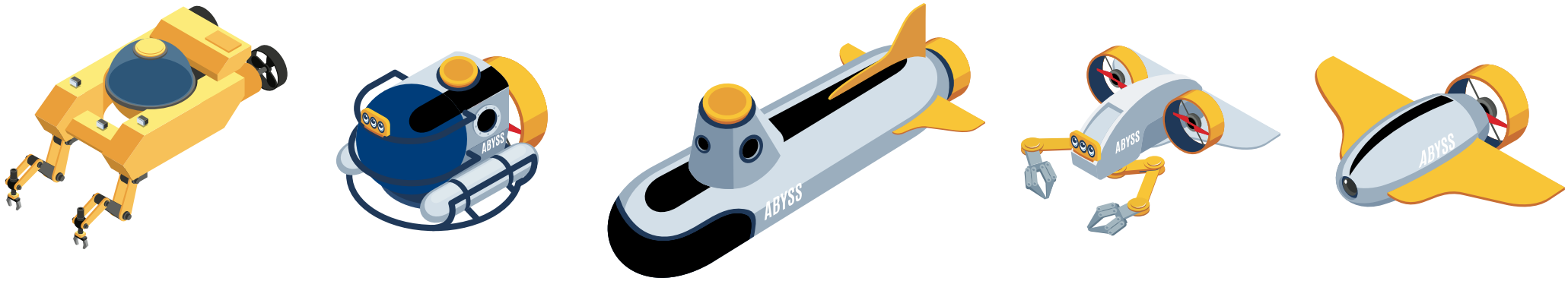
Developers in Scrum must engage in discovery and are part of the creative process. They explore the unknown.

This adventure contains various activities encouraging interplay. These focus on themes like failure, safety, ownership, complexity, technical debt, professionalism, integration, and quality. After all, an unsafe Development Culture results in unsafe (volatile, fragile) products.

The biggest challenge is turning that *I* into a *WE*, which begins by respecting people first for being capable *individuals*.

During this adventure, we refer to participants as travelers, divers or crew.

- Explore what's below the surface;
- Surface technical debt;
- Reveal the error culture;
- Expand accountability over quality and delivery;
- Try patterns for deployment and delivery strategies;
- Try patterns for cultivating a 'whole-team' mindset;



Dear crew of "the Abyss",

Our mission today is to venture deep within developer territory; explore strange new ways; seek out technical debt and error culture, and push the boundaries of what is known and what is possible.

When we seek out the unknown, we will find complex things that challenge us. That's how we come to rely on each other, and trust each other. For that trust will be like oxygen. Our commitment to supporting each other will be our hull. The stronger it is, the more we can cope with pressures and dive deeper.

Rig for dive. Clear the bridge.

Dive, dive, dive

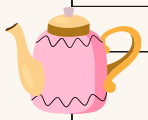
What's your cup of tea?

(A)TDD

CI / CD

Clean Code

DevOps



A/B Testing

Unit Testing

Refactoring

Application Telemetry

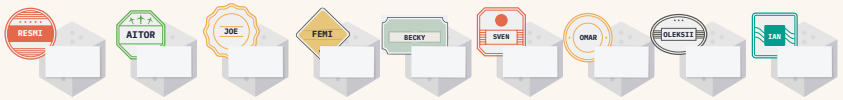
Collective Ownership

Swarming / Pairing / Pair Programming

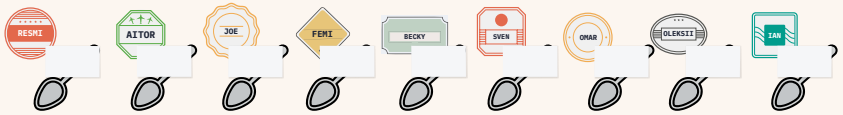
Kanban

QA

Sugar with the tea?



Let's stir it!



Mad Tea Party

As we submerge, we can enjoy a cup of tea. What's your cup of tea? You might recall our Mad Tea Party back at Basecamp. In this variant, each crewmember can share their cup of tea concerning professional developer practices. As a guide, you can prepare various cups of tea containing a specific professional developer practice, such as Clean Code, Pair Programming, Refactoring, etc.

1 minute	Each crew member places their token in a cup with a practice they are most familiar with.
5 minutes	Let's add sugar: Each crew member writes down questions (sugar) they have about the various practices and places the questions in the corresponding cups.
5 minutes	Let's stir it: 1. Pick a cup with your token in it. 2. Team up when there are more tokens in one cup. 3. Run through the questions (sugar) in the cup. 4. Write down the answers to the questions in the cup. 5. Fold the question and answer together. 6. When there is no sugar in your cup, write down as many facts you know about that practice.
2 minutes (per crewmember)	All together, each crew member selects one cup. This may be their own cup, someone else's, or even one without any tokens. Together review the cup (two minutes per turn).

"Take some more tea," the March Hare said to Alice, very earnestly.
"I've had nothing yet," Alice replied in an offended tone, "so I can't take more."
"You mean you can't take less," said the Hatter: "it's very easy to take more than nothing."
"Alice, are you familiar with this special kind of TDD?" the March Hare continues.
"Tea Dee Dee? now you completely lost me" Alice sighs.
"Do you 'CI' Alice?" asked the Hatter.
"Don't you mean 'see me'?!" Alice replies, "Yes of course I do see you".
"Well, how about 'CD'?" the Hatter continues.
"No, who is this Dee?!" Alice frowns.

Red culture

When you fail, you lose.

Errors are failures that should be avoided at all costs. Bugs are mistakes that should instantly be corrected. They are negative.

It's important to find out who made the mistake. The one who broke it must fix it.

Developers need permission for refactoring their code or performing technical updates. They can do this when 'there is time' for that.

The one who is most experienced should to the most complex work. Thats the most efficient way.

Blue culture

You can't win if you don't fail

We welcome the discovery of defects. We welcome small, fast and frequent changes enabling us to fail fast and resolve fast.

We are **blameless** about defects. We collectively resolve them so we can all learn from them.

Developers take collective code ownership. They refactor continuously. They update daily.

We often pair up and improve each others code. We learn, share knowledge, and reduce individual dependencies.

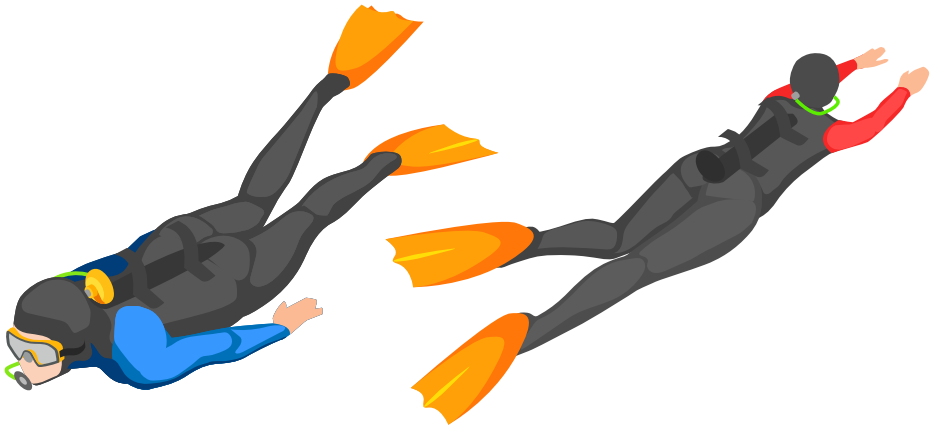
Error Culture

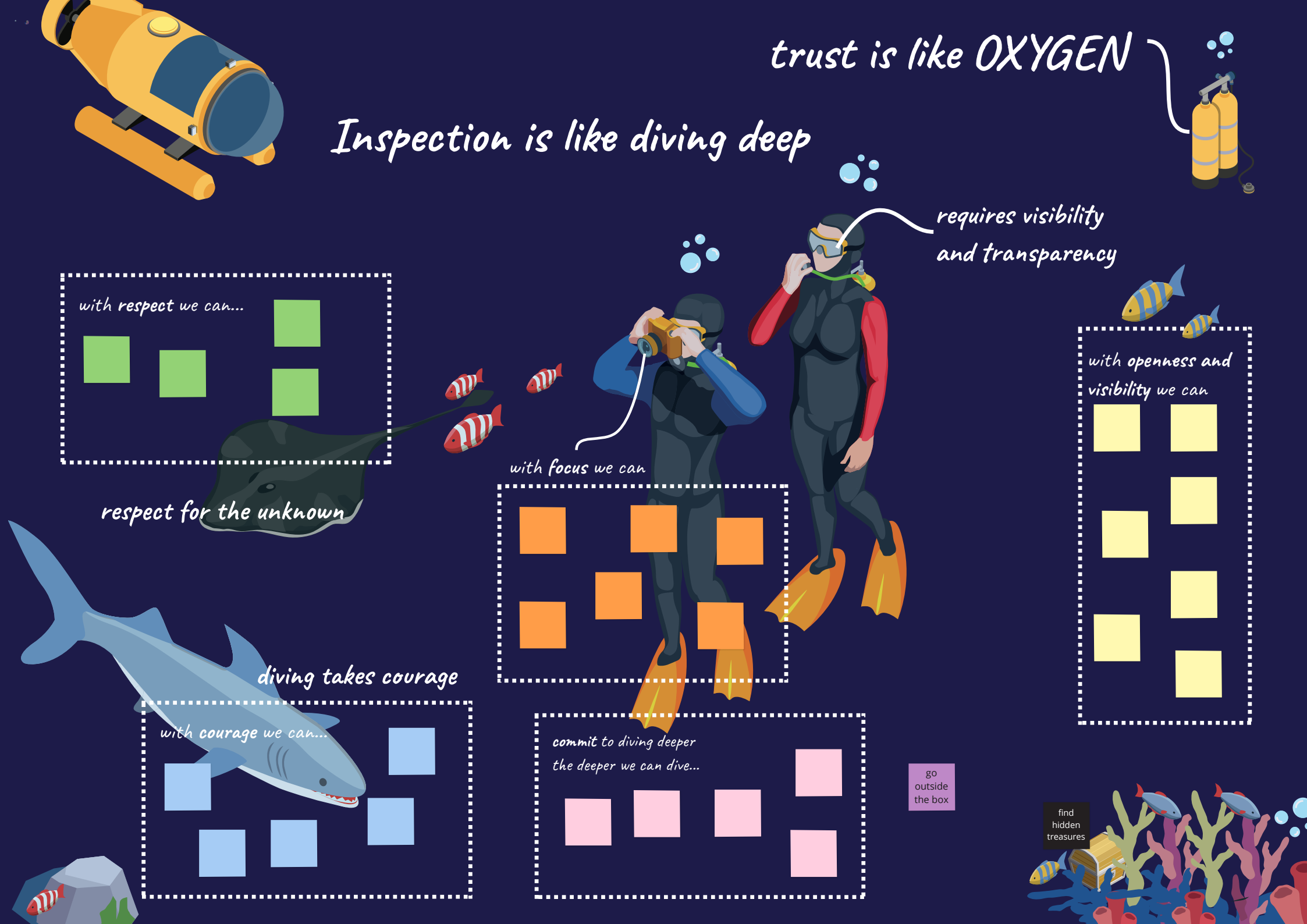
One who makes no mistakes makes nothing. An unsafe Development Culture results in unsafe (volatile, fragile) systems. In a way, the culture is reflected in the code.

“Error Culture: How mistakes are handled impacts an organization’s ability to innovate. If people feel that errors are something negative and try to avoid them, they might be much less likely to take a risk and try something new. Instead, encouragement for experimentation and learning is important while at the same time considering how to tame risks and decrease the impact of failures.”
– Scrum.org Developer Glossary.

Let’s compare the two cultures of Submarine Blue and Submarine Red. Which ship/crew would you prefer to join?

10 minutes Invite the crew to pair up. Together, reflect on the Blue and Red cultures.




trust is like OXYGEN
Inspection is like diving deep
requires visibility
and transparency
with respect we can...
respect for the unknown
with focus we can
with openness and
visibility we can
diving takes courage
with courage we can...
commit to diving deeper
the deeper we can dive...
go
outside
the box
find
hidden
treasures

Values Deep Dive

“When these values are embodied by the Scrum Team and the people they work with, the empirical Scrum pillars of transparency, inspection, and adaptation come to life building trust.” – The Scrum Guide.

Inspection with courage is like deep-diving, whereas trust is like OXYGEN. We must trust each other and the gear. It requires diving masks/goggles for clear visibility.

Diving deeper requires a real commitment. It takes courage to face the dangers of the deep. There must be respect for the unknown and each other. Open communication is lifesaving.

“Trust is a bit like the oxygen we breathe; it sustains, it brings things to life, but without it... we are left gasping. A Scrum team can dive deeper with trust to sustain the team.” – John Albrecht

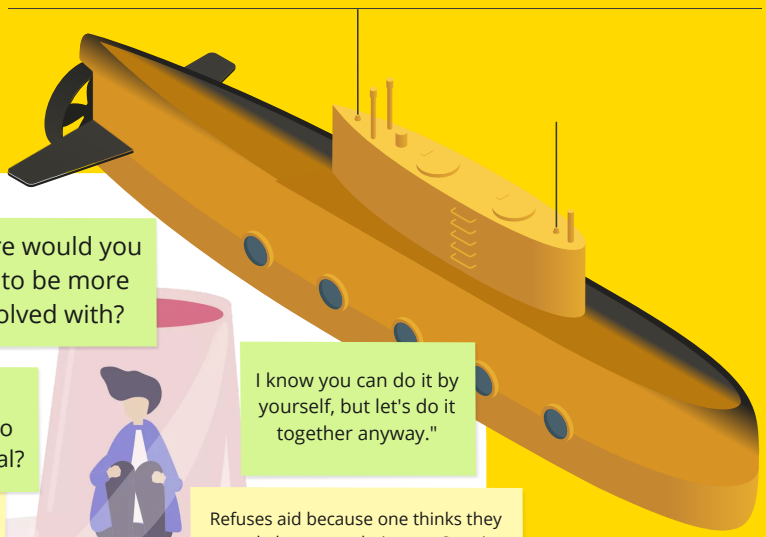
5 minutes

Divide the crew across five breakouts for each of the Scrum Values:

1. With respect we can...
2. With focus we can...
3. With openness we can...
4. With courage we can...
5. With a commitment to diving deeper, the deeper we can dive...

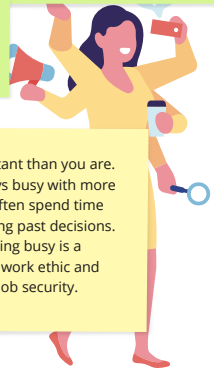
10 minutes

Each breakout presents its key take-aways to the main group.



Be patient with others like how you want others to be patient with you.

Slowing down is a good way to speed up.

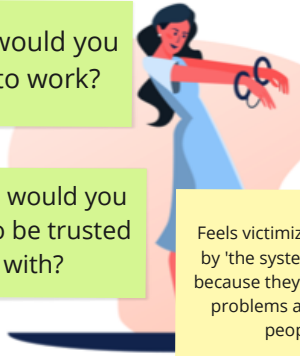


Their work is more important than you are. That is why they are always busy with more 'important stuff'. They often spend time backtracking and discussing past decisions. They assume that being busy is a demonstration of good work ethic and provides them with job security.

Busybee

what would happen if things go your way?

How would you like to work?



What would you like to be trusted with?

Feels victimized and disempowered by 'the system'. They won't change because they are tied down. All their problems are the result of other people's decisions.

Prisoner

where would you like to be more involved with?

What does it mean for you to be a professional?

I know you can do it by yourself, but let's do it together anyway."

Refuses aid because one thinks they can do better on their own. Says is not responsible to avoid accountability. They stay inside their comfort zones doing what they know and do best.

Soloist

Be surprised how much you can learn from those who know less than you.

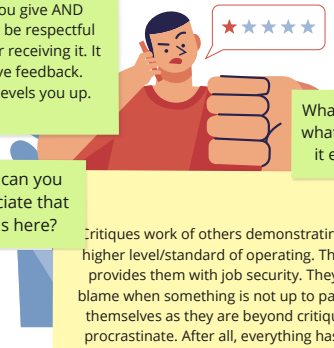
I trust your way will work, but let's try it someone else's way so we can learn.

Dismisses and downplays ideas of others and presents their own as better/smarter.

Be patient with others like how you want others to be patient with you.

Smartass

Be thankful when you give AND receive feedback and be respectful to the person giving or receiving it. It takes courage to give feedback. Feedback is a gift. It levels you up.



What do you like? what would make it even better?

What can you appreciate that got us here?

Critiques work of others demonstrating they are on a higher level/standard of operating. They assume this provides them with job security. They know who to blame when something is not up to par (and it's never themselves as they are beyond critique). They often procrastinate. After all, everything has to be perfect.

Perfectionist

I trust your way will work, but let's try it someone else's way so we can learn

I know you can do it by yourself, but let's do it together anyway."

These ones step in to save the day by taking on the most complex work and critical challenges. They keep essential know-how to themselves as it gives them job security and a strong negotiation position at their next performance review. They are generally impatient and often make colleagues feel inferior.

Hero

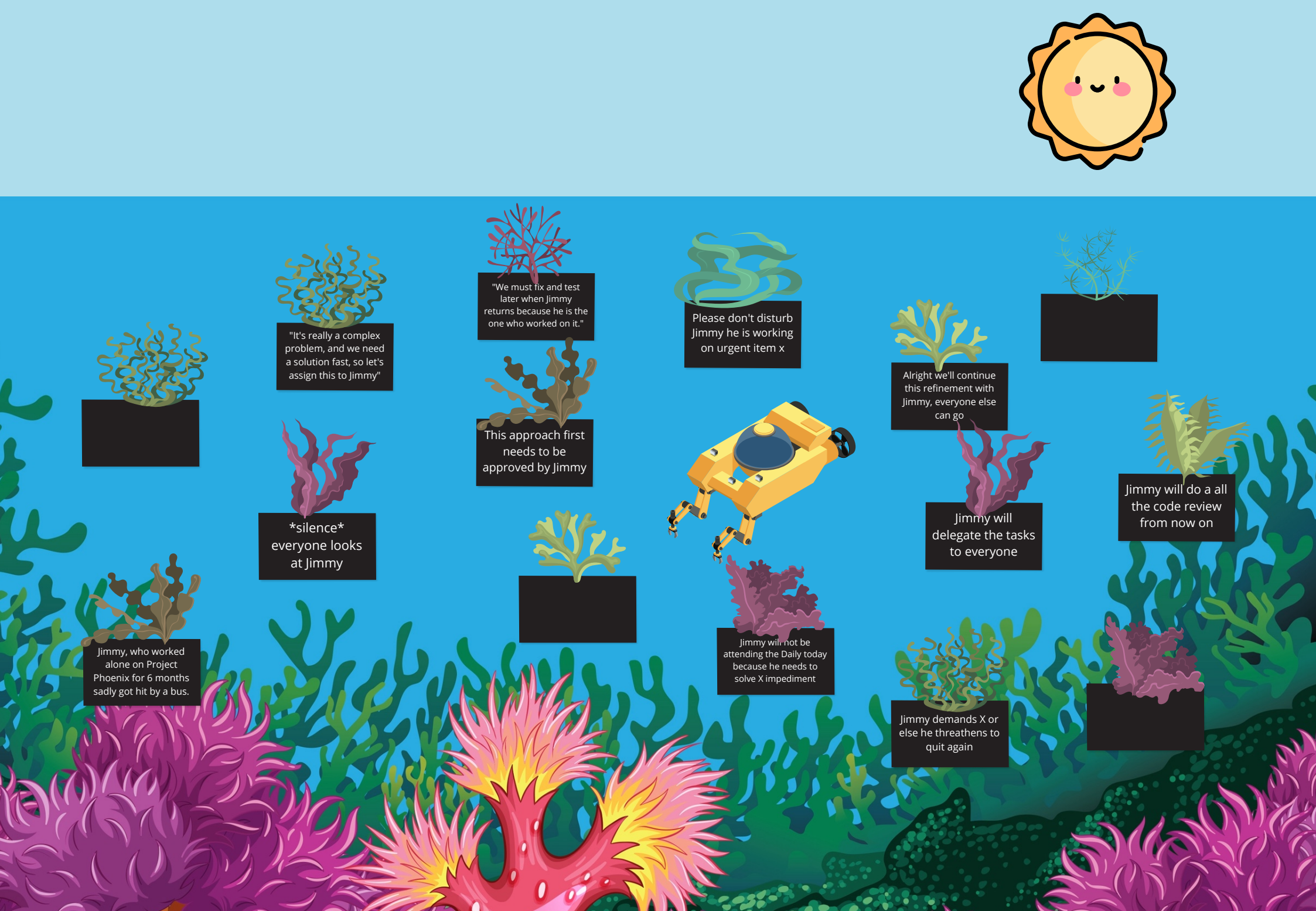
We all live in a Yellow Submarine

There isn't enough space on our little submarine for BIG egos. We all have egos. Yes, coaches too. That's not a bad thing. Good intentions drive our egos, yet they may result in negative outcomes from time to time. Our egos may shield us from (or mask) our insecurities.

To make it safe to play: share that these egos are stereotypes, not archetypes. Don't label individuals this way! We may recognize ourselves in these egos as our pitfalls.

5 minutes	Invite the crew to map the yellow sticky notes (containing descriptions of egos) to what they think the ego corresponds with.
5 minutes	Verify if the mapping is done correctly (according to what is shown to the left). Now ask the crew what positive intentions may be driving those egos.
5 minutes	Invite the crew to write coaching questions or encouraging statements to the various egos using green sticky notes.

Although one may consider someone to act disrespectfully, displaying negative characteristics of these egos, it would be equally disrespectful to label them using these stereotypical names. Human beings are far more complex and are ultimately driven by good intentions. That means you should be open to sharing your observations when behavior leads to negative outcomes.


"It's really a complex problem, and we need a solution fast, so let's assign this to Jimmy"
"We must fix and test later when Jimmy returns because he is the one who worked on it."
Please don't disturb Jimmy he is working on urgent item x
Alright we'll continue this refinement with Jimmy, everyone else can go
This approach first needs to be approved by Jimmy
silence everyone looks at Jimmy
Jimmy will delegate the tasks to everyone
Jimmy will do a all the code review from now on
Jimmy, who worked alone on Project Phoenix for 6 months sadly got hit by a bus.
Jimmy will not be attending the Daily today because he needs to solve X impediment
Jimmy demands X or else he threathens to quit again

Remove the Shade

Stepping aside, so others can step in/up. The Road to Mastery (R2M) uses guiding principles from Training from the Back of the Room (TBR). One of those principles is for the trainer to step aside to let the learners learn. Similarly, leaders can step aside to let others lead. Managers can step aside to let others manage. Metaphorically, we step aside to remove the shadow we cast on others. This allows others to step into the light, shine, and grow.

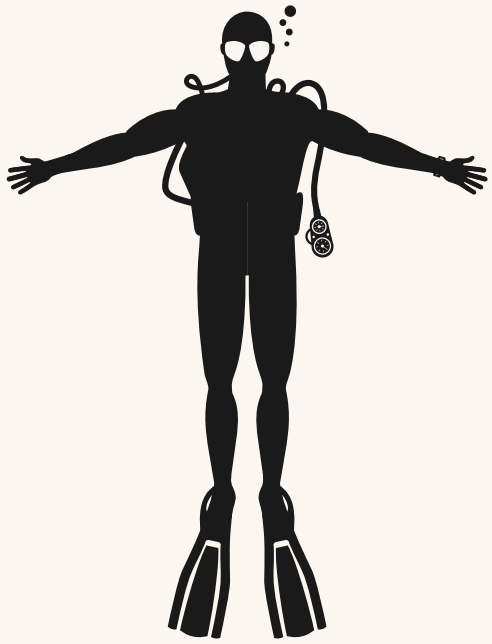
7,5 minutes	Can the crew name at least ten ways someone can *overshadow* (and limit) individuals from mastering abilities and increasing their potential? Can they differentiate between shadow cast from outside or within the team?
7,5 minutes	According to the group, what are the approaches to removing the shade?

Removing shade begins with you. A Scrum Master may have the drive to help. But rather than inflicting help, you can consider certain situations as learning opportunities. What will they do about it? You may trust that your way will work, and what about their way?

Attribution: This play is based on “Remove the Shade”, a Scrum Pattern by Dina Friis and Gertrud Bjørnvig.

Survive this Dive!

- Imagine you're about to take a plunge into cold water
- Ready? *jump*
- Breathe in out deeply and calmly three times
- Hold your breath for 10 seconds after a sharp inhale
- Simulate different swim strokes (Butterfly, Breaststroke, Backstroke)
- While on a chair, paddle your feet in front of you (as if you are wearing flippers)
- Simulate various diving signs
- Thumbs up to signal the end of this dive



Checkpoint 1

We started this dive to learn what it takes to be part of a crew. Our trust allows us to go deeper. As we descend, we (re-) discover the importance of light to our ecosystem.

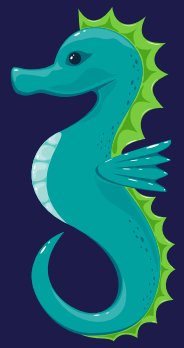
5 minutes

1. Did the previous activities reveal anything about yourself?

We're going to encounter some strange creatures on our way down. Let's trust our submarines are well-engineered and do not contain Technical Debt as we venture into the technical depths.

- "Crew, search from starboard beam to bow."
    - "Clear all around."
- "Search thirty degrees on each side of the bow."
    - "Doubtful contact... bearing thuh-ree fi-yiv ze-ro."
- "Pick up target bearing ze-ro eight six."
    - "Target bearing ze-ro eight fo-wer, slow heavy screws."
- "Stay on target bearing two fo-wer seven."
    - "Target bearing two fo-wer seven, light screws speeding up."
- "Give me a mark on bearing two fo-wer seven."
    - "Mark!"
- "Monitor noise."
    -"High noise level bearing one six one!"

The 4 Sea Horsemen of Technical ~~Debt~~ Depth

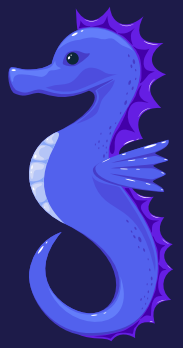


Fragility

The software **easily breaks** in many places due to a single change. There are unknown relationships between seemingly unrelated areas. Fixes cause more problems. Performance is easily disrupted or interrupted. There are many unexpected incidents.

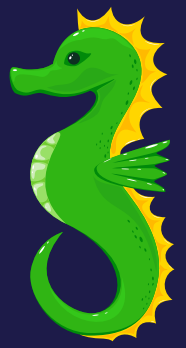
Obscurity

The code is **hard to understand**. There is needless complexity and needless repetition. It is not clear why or how something works (or doesn't). There is reluctancy to make changes in existing code and tendency to want to start over.



Rigidity

The software is **difficult to change**. A small change causes a cascade of subsequent changes. Preference for dirty work arounds. Preference to ignore the problem because it's too risky to solve.



Immobility

You **cannot reuse (**parts of) the code or migrate it because it involves risks and high effort. The code has many of dependencies. Difficult to update the code because it causes a cascade of subsequent updates.

The Four (Sea)Horsemen

Time to talk about the giant evil monster in our Product Development universe. Is it our Cthulhu, Morgoth, Chimera, or Frankenstein? Or is it the Grim Reaper that will one day come knocking on the door of every product made?

"Technical Debt: the typically unpredictable overhead of maintaining the product, often caused by less than ideal design decisions, contributing to the total cost of ownership. May exist unintentionally in the Increment or introduced purposefully to realize value earlier."
— Scrum.org Developer Glossary

Face its dread today? or its calamity tomorrow? Technical Debt is attributed to poor design, architecture, or managerial decisions. Yet it's bigger than all those who contributed to the product. When the product lives in a complex and volatile market, Technical Debt involves unknown unknowns (scary huh!) and continues to emerge even when the product isn't. And that makes it so hard to take collective ownership of it as not a single person can predict, understand, or grasp it completely.

When ignored, don't be surprised when these four Harbingers of Product Death, the (Sea)Horsemen of the Apocalypse, come knocking:

5 minutes Invite the crew to map the four descriptions to their corresponding seahorses.

The trouble with Technical Debt is that, as it grows, it *obscures*. Obscurity is the nemesis of Transparency and the first of our horsemen. It results in a scattered and distorted view, poorer decisions, and the draining of value. For example, something might look like a cute little update, but the tweak might reveal a dependency nightmare.

It is *fragile* when the software/product easily breaks in many places due to a single change. There are unknown relationships between seemingly unrelated areas. Fixes cause more problems. Performance is easily disrupted or interrupted. There are many unexpected incidents.

It becomes *immobile* when you cannot reuse parts of the code because of the risks and high effort it will require. The code has too many dependencies. It becomes difficult to update the code because it causes a cascade of subsequent updates.

Now the software becomes too difficult to change. It grows *rigid*. Everything slows down. A small change causes a cascade of subsequent changes. The developers may develop a preference for dirty workarounds. They ignore the problem because it's too risky to solve, and they are under pressure to pick up speed.


1
2
3
4
5
ABYSS

Something's Fishy

The Scrum Team develops quality increments that adhere to the Definition of Done. That does not mean the product will always be in a perfect state. Unexpected things do occur. When something's fishy in the product this may be referred to as a defect, damage, or bug. How should these fishy encounters ideally be reported? It's up to the crew!

8 minutes

Part 1: Damage Report. Invite the crew to:

1. Individually write 5 essential ingredients for a Damage Report. (2 minutes)
2. Pair up. Merge your list with your partner. Narrow the list down to 5. (2 minutes)
3. Merge two pairs. Merge the lists and narrow it down to 5. (2 minutes)
4. With the main group, merge the list and narrow it down to 5. (2 minutes)

8 minutes

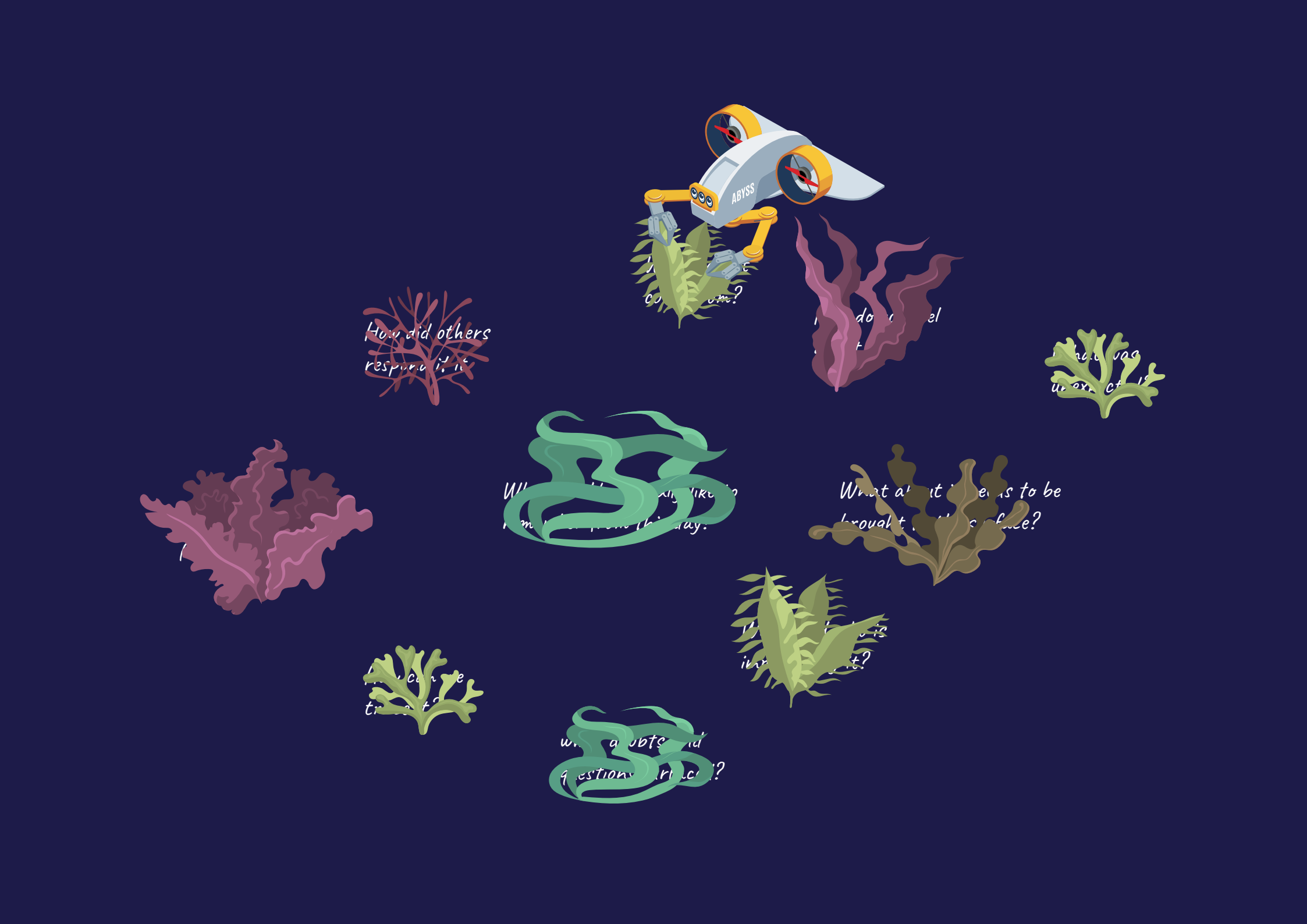
Part 2: Response Team. Invite the crew to:

1. Individually write 5 good practices for finding and fixing defects. (2 minutes)
2. Pair up. Merge your list with your partner. Narrow the list down to 5. (2 minutes)
3. Merge two pairs. Merge the lists and narrow it down to 5. (2 minutes)
4. With the main group, merge the list and narrow it down to 5. (2 minutes)

8 minutes

Part 3: Prevention. Prevention is better than responding. Invite the crew to:

1. Individually write 5 good practices for preventing defects from occurring. (2 minutes)
2. Pair up. Merge your list with your partner. Narrow the list down to 5. (2 minutes)
3. Merge two pairs. Merge the lists and narrow it down to 5. (2 minutes
4. With the main group, merge the list and narrow it down to 5. (2 minutes)


ABYSS
How did others

Unexpected Encounters

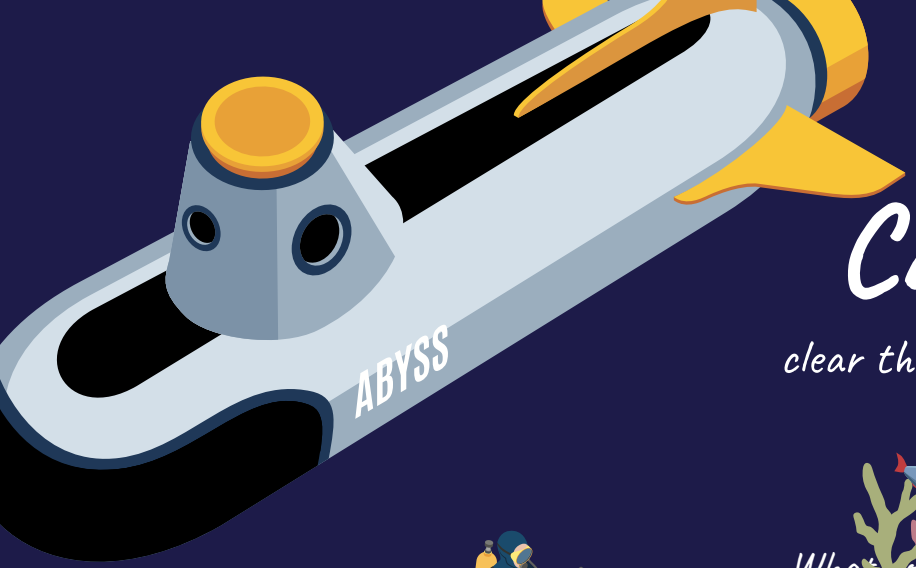
Working in a complex environment, Developers will encounter unexpected things. We can inspect them, learn from them and adapt to them. Unexpected Encounters is a cheat sheet of optional questions the development crew may ask themselves to inspect the unexpected.

1. What was unexpected about this encounter?
2. Where did it come from?
3. How do you feel about it?
4. What about it needs to be brought to the surface?
5. What and who is impacted by it?
6. What doubts and questions surfaced?
7. How can we trace it?
8. What can we learn from it?
9. How did you and others respond to it?
10. What would you really like to remember from this encounter?

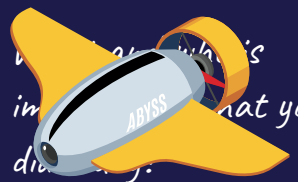
5 minutes	To prep: As a guide, prepare the set of questions and hide them by covering them. Share and explain the unexpected encounter with the development crew.
25 minutes	Ask the crew to reveal a question. They may then answer it and invite other crew members to answer it. Optional: record the answers.

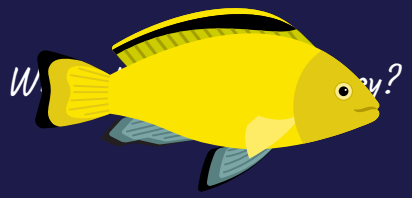
Captain's Log

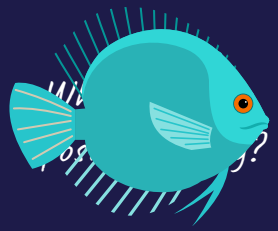
clear the items and record your answers

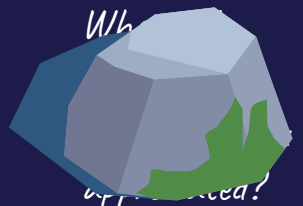


today









Captain's Log

Empiricism is the theory that human knowledge comes predominantly from experiences gathered through our senses. We study what is happening by documenting and theorizing about what might, should, or could happen. Creating logs and noting down observations when it is still fresh can significantly improve inspection and decision-making.

I encountered teams that spent enormous amounts of time and energy on (technical) requirements and design specifications defining what might be needed. They did not, however, record what was actually developed (or how), nor how it was being used. Needless to say, all those by-products proved to be useless, misleading, and wasteful.

Like Unexpected Encounters, the Captain Log can record notable observations. These can prove helpful during Sprint Retrospectives, for example.

Naturally, it is entirely up to the crew to determine if and when to record their logs. Here are prompts (cheat sheet) that may be helpful when recording log entries.

1. What gave you positive energy today?
2. What do you remember feeling today?
3. What was unexpected about today?
4. What and who is impacted by what you did today?
5. What's missing today?
6. What confused you today?
7. What drained your energy today?
8. What did today bring to the surface?
9. What would you really like to remember from today?
10. What did somebody do today that you appreciated?

10 minutes

Invite the crew to individually record their log using three to five questions (prompts). This may be an audio or video recording or a written log. Audio or video logs are generally faster, simpler, and more personal to make.


ABYSS

Product Telemetry (Radar)

“Radar, search from one ze-ro ze-ro to one fi-yiv ze-ro.” - “Radar, contact, bearing one fo-wer ze-ro, range... “

Understanding how a product is used and how it is performing is key for making better decisions on where to invest next. Product (or Application) Telemetry makes evidence visible.

“Application Telemetry: Understanding how a product is used is a key factor for taking better decisions on where to invest. Application Telemetry can provide some insights to increase this understanding by showing usage statistics, performance parameters, user workflows and other relevant information.” - Scrum.org Professional Scrum Developer Glossary.

15 minutes

Invite the crew to form triads or quads. They may imagine and visualize (draw) their radar. What might the radar track and show in context to the usage, performance, and health of the product?

A radar can prove vital as it can enable timely decision-making. You don’t just want to have a dashboard showing damage reports. You want to see what’s happening and coming your way.

Product Telemetry may show both “leading” and “lagging” indicators. A “leading” indicator informs us how to produce desired results (outcomes), and a “lagging” indicator measures current production, in and output, and performance. In short: a leading indicator looks forward to future outcomes and events. A lagging indicator looks back at whether the intended result was achieved.

10 minutes

Invite the crew to review the radars. What indicators might qualify as “leading” and which as “lagging”? Is there a relationship between indicators? Might a change in one lead to a change in another

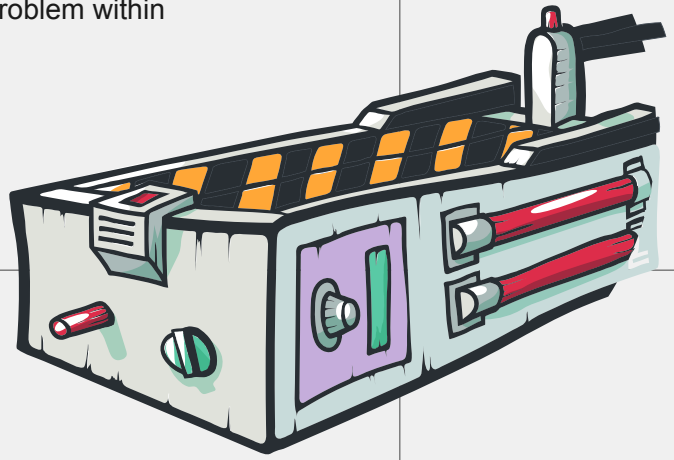
Ghost Trapper!

1. Find!

Find an example of a harmful problem within the system.

2. Capture!

Capture it by developing a shared understanding of it through exchanging perspectives. Visualize it on a board. Inspect closely and notice details.



3. Explain!

After you have visually and mentally captured it, write a paragraph (or show and tell another colleague) how it is complex.
Can you explain it to an 85 or 5 year old?

4. Wonder!

What new insights were revealed?
What new ideas and questions do you have?
Are there ways to make it harmless or less harmful?

Ghost Trapper

We're once again encountering strange phenomena. These appear to be underwater boos roaming the technical depths. Are they real or hallucinations? As we ascend to a shallower depth, the pressure changes. While diving, narcosis may cause a (reversible) alteration in consciousness that occurs at depth.

Technical Debt is a hard challenge for Scrum Team to manage. If only teams could apply something like a Ghost Buster contraption to suck in and capture those malicious spirits.

Ghost Trapper involves four steps taken by a team of (ideally) cross-functional specialists: find, capture, explain, and wonder.
The crew may skip this play, yet it may work well as a "Call to Adventure" where they can trap Ghosts with their Scrum Team.
(This play can work well combined with 'Tiny Monsters' (from the Valley of Values) to visualize the Ghosts.

< 15 minutes

1. Find an example of the complex harmful problem within the system.

< 15 minutes

1. Take some time to look carefully.
2. Capture it by developing a shared understanding of it by exchanging perspectives.
3. Visualize it on a board.
4. Inspect closely and notice details.

< 15 minutes

1. After you have visually and mentally captured it, write a paragraph (or show and tell another colleague) about how it is complex.
2. How would you explain it to an 85 or 5-year-old?

< 15 minutes

1. What new insights are revealed?
2. What new ideas and questions do you have?
3. Are there ways to make it harmless or less harmful?

Attribution: Ghost Trapper is based on "Slow Complexity Capture" a thinking routine by HARVARD GRADUATE SCHOOL OF EDUCATION: Project Zero.

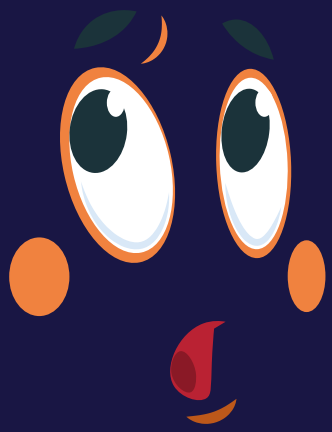
Stretch Break: Funny faces & face massage

Watch out for signs of compression sickness or oxygen deficiency!

Make funny faces!
(You can do this with camera off)
This helps you to relax your face muscles.

Tight neck and shoulder muscles often limit the circulation to the brain, which consequently diminishes cognition, memory and concentration.

So, let's follow this up with a short shoulder rub, head-roll and face massage.



Checkpoint 2

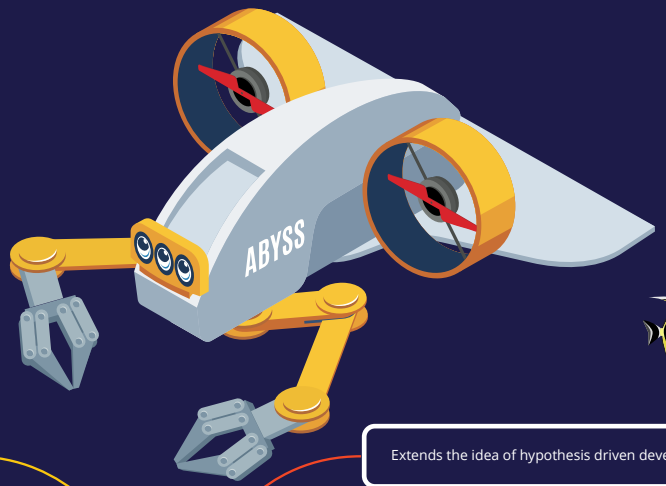
Boy, we're in deep! We've explored the unknown and encountered strange phenomena. We've checked and tested our systems and gained more confidence in our vessels and crew.

5 minutes

1. Did the previous activities reveal anything about the quality culture of your product?
2. How can you share your learnings with your (development)organization?

It's time to start heading back up. But note! we're not quite there yet. There's more waiting for us on the surface.

- "Make all preparations for surfacing."
- "Maneuvering, on surfacing answer bells on thuh-ree main engines; put one main engine on charge." - "Secure the ventilation. Shut bulkhead flappers."
- "All compartments report in turn that bulkhead flappers are shut."
- "The forward engine room also reports ventilation secured."
- "Ready to surface in all respects."



- Exploratory Testing
- Unit Testing
- A/B Testing
- Smoke Testing
- Functional Testing
- Integration Testing
- Acceptence Testing

- Extends the idea of hypothesis driven development by evaluating two or more different implementations.
- Minimal set of tests run on each build to test and assess wether or not the system overall is stable.
- A type of testing which is not created in advance, but testers check system on the fly. This amplifies the personal freedom, responsibility. It relies on human interaction.
- Test whether the behavior acts in accordance to what is expected and desired as it responds to interactions.
- low-level technical test focusing on small parts of a software system that can be executed fast and in isolation
- Determine if the requirements of a specification are met to an acceptable standard.
- Test whether various functionalities behave as expected and work well together. It can be used to test a complete user flow or other sequence of behaviors and interactions.

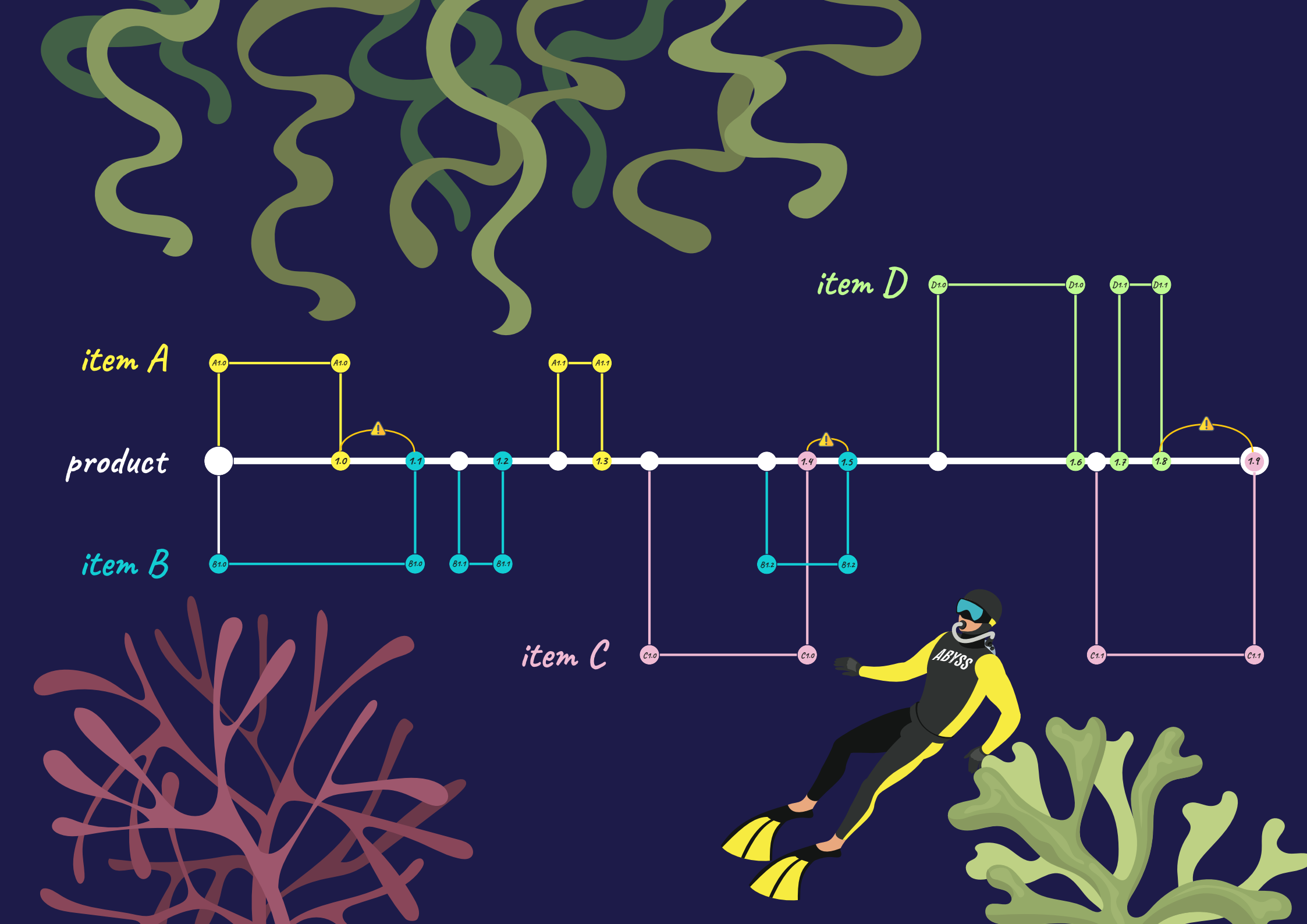
Connect the Two

The Developers themselves are accountable for instilling quality. They do so, in part, by adhering to the Definition of Done. How this is done may vary. Although Scrum Developers can work with others outside the team, they remain accountable. They instill the quality of new increments and ensure that the sum of previous increments (the product) continues to adhere to the quality standards.

In this play, our crew explores seven approaches to testing. There are more approaches or subsets that the crew can expand upon.

Let's see if our crew can (re)connect these types of testing to their corresponding definitions.

5 minutes	As a guide, rewire the connections so that they are incorrect. Invite crew members to take turns rewiring a connection. A connection may be rewired twice if a crew member believes the previous connection is incorrect.
1 minute	Aks the crew: which of these tests is not to be automated?


item A
item B
item C
item D
product
A1.0
A1.0
A1.1
A1.1
B1.0
B1.0
B1.1
B1.1
B1.2
B1.2
C1.0
C1.0
C1.1
C1.1
D1.0
D1.0
D1.1
D1.1
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
ABYSS

Branching and Merging

Developers manage incremental development with continuous integrations through branching and merging strategies with release pipelines.

Branching: Creating a logical or physical copy of code within a version control system so that this copy can be changed in isolation.

Merging: When the work is completed and tested, the code from that branch merges into the main codebase.

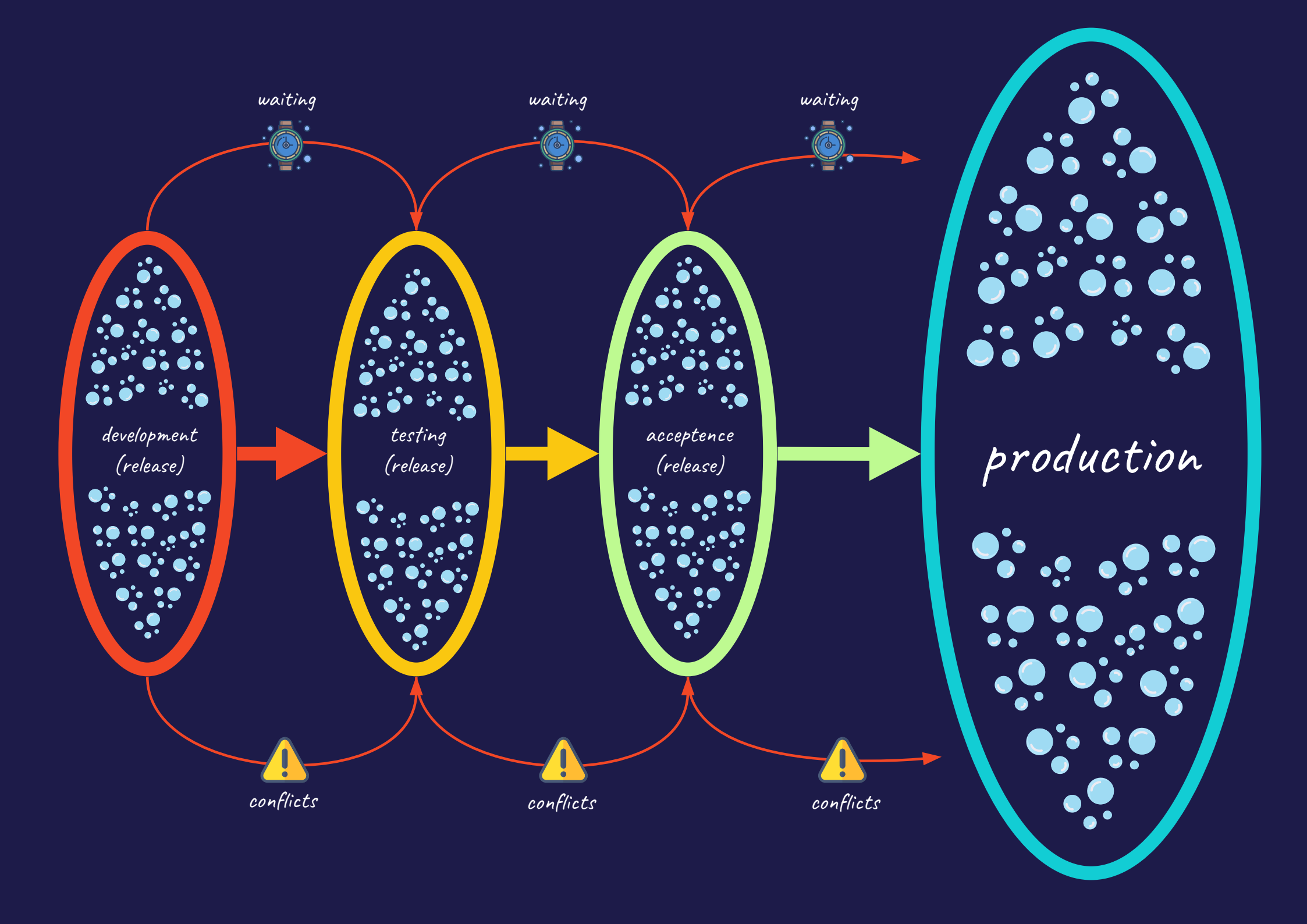
Release-Pipelines: Automating the steps from code commit into version control to production delivery increases both speed and reliability of deployment.

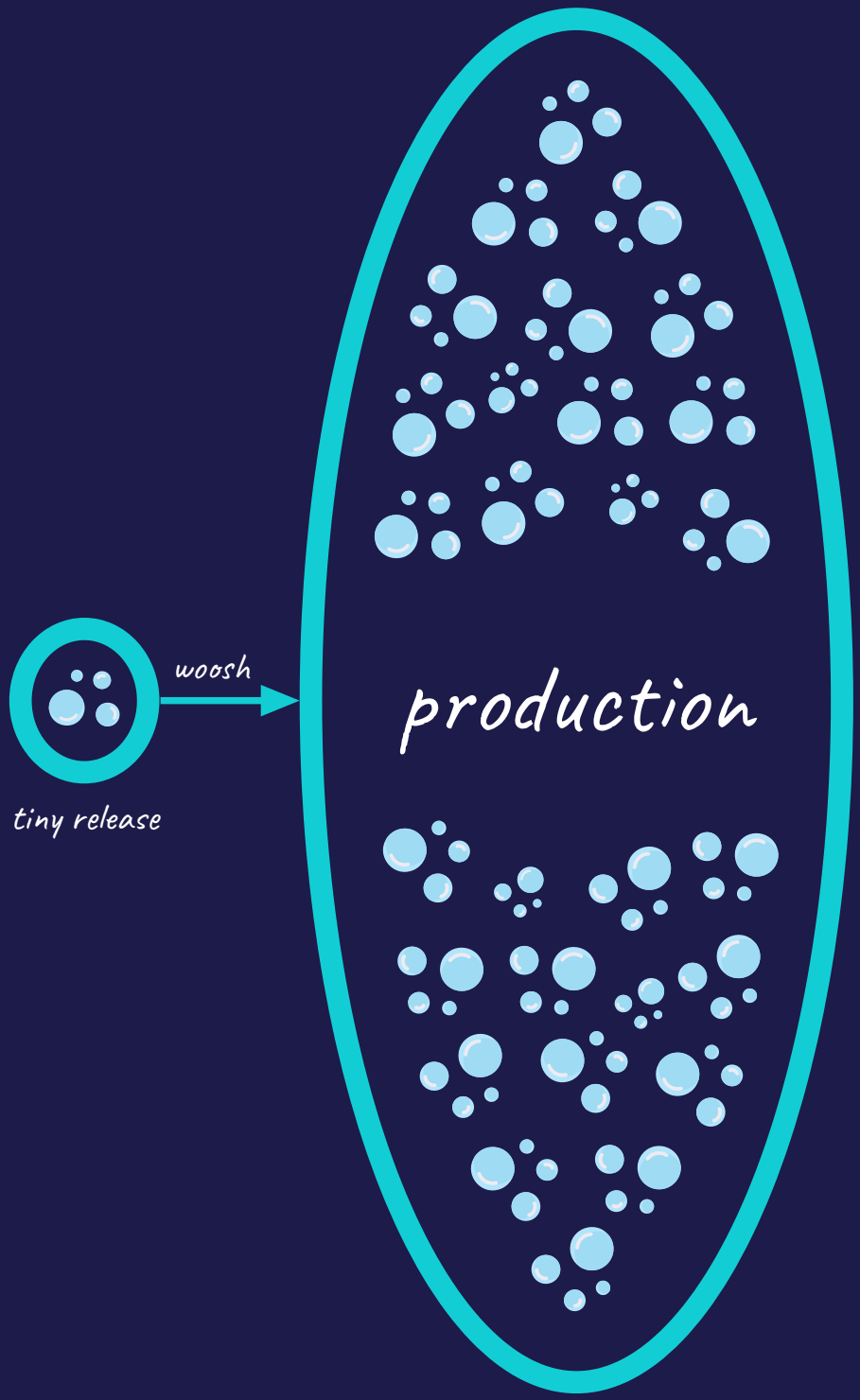
Codebases can get rather tangled and intertwined. Branching and Merging are a part of (incremental) version control for managing codebases. When developers start a new branch, the version control system creates a copy of the codebase. Changes to the branch won't affect other branches. Once the developers complete the work, they may merge their work into the central code base and close their branch.

There may be multiple strategies for branching and Merging. Do branches represent tasks, items, features, or releases? Version control exists, in part, to benefit transparency and quality. By no means should it become a source of obscurity and fragility! As a general rule of thumb, simplicity is a good strategy for engaging complexity. This means that, in general, you may want only a few short-lived open branches. You don't want to leave 'm hanging. When there are many unfinished branches, and if they exist for a long time, this will introduce obscurity. The longer a branch is open, the higher the risk of conflicts when merging, which will be more difficult to resolve.

15 minutes

Invite crewmembers to draw and share what their branching strategy looks like.
Break into pairs or triads if and when needed.


waiting
waiting
waiting
development
(release)
testing
(release)
acceptence
(release)
production
conflicts
conflicts
conflicts


woosh
tiny release
production

automated alerting

active monitoring

frequent micro releases

team is trusted with autonomy to decide over releasing

Blue/Green Deployment

gradual rollout where only a small percentage gets the new functionality first which then gradually increases.

clean code

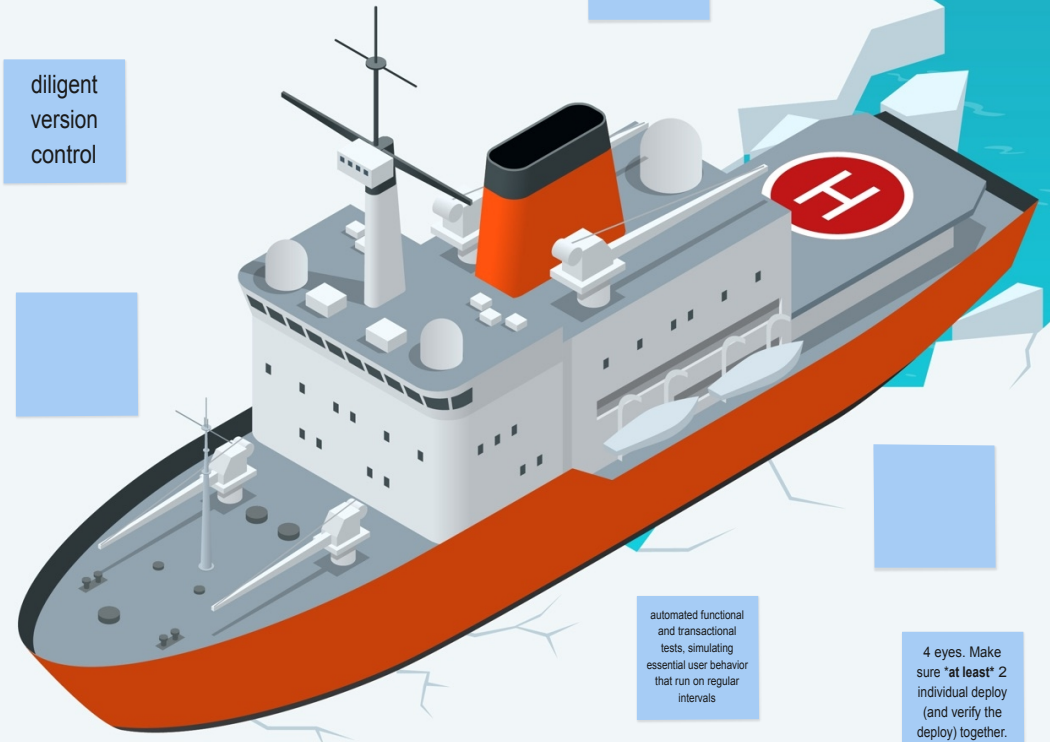
Mob/Swarm to resolve any issues regarding deployment

diligent version control

daily team inspections of all releases that day

feature toggles

code reviews



automated functional and transactional tests, simulating essential user behavior that run on regular intervals

back-up prior to deployments

Rubber Chicken: only one person can deploy at a time. Grab the rubber chicken if you want to deploy. Only the one with the Rubber Chicken can deploy.

limit the amount of work in progress (finish before starting)

4 eyes. Make sure ***at least*** 2 individual deploy (and verify the deploy) together.

Breaking Through

The crew of the Abyss made it safely to the surface. Our mission still needs to be completed. We are docking with Ice Dragon, a Nuclear Icebreaker Vessel, which will take us to the station Arctic Observatory 4.

Deploying hot to production sounds risky (it is). That is why development organizations apply pre-production environments such as staging, test, and acceptance environments. One such strategy is DTAP.

Yet, this also introduces complexity and does not come without risk. More often than not, this treatment can prove way worse than the disease.

- Discrepancies between environments harm transparency and increase the chances of making mistakes.
- The setup provides a false sense of security.
- Developers may grow more complacent and thus less diligent over quality (“It worked on my machine” / “I am not a tester” / “It’s
- only on test” / “I’ll check it later” / “That’s not my work”, etc).
- It diminishes the sense of collective ownership.
- Dependencies with yet undeployed code emerge and grow over time.
- Data inconsistencies emerge and grow over time.
- Releasing becomes more complicated, and time-consuming.
- The art of releasing becomes a specialized competence only a few master. It may become a silo.
- It becomes increasingly complicated to find the root cause of issues.
- It significantly increases time-to-market.
- It significantly increases the cost of ownership of the product.

The sunk-cost fallacy kicks in when the development organization learns that the DTAP is not the silver bullet. Fragility, Rigidity, Obscurity, and Immobility are roaming around. The development organizations fight them with even more policies, rules, and tech, which only fuel them.

15 minutes

Divide the crew into triads or quads. Imagine if the development organization had no option but to release hot to production. What ways to cool things down, to break safely through to production, that will not contribute to Fragility, Rigidity, Obscurity, and Immobility?


ARCTIC
ARCTIC

Checkpoint 3

Welcome to Arctic Observatory 4. Getting here took courage and commitment.

You made your way through dangerous technical depths and broke through rigid ice sheets. It's freezing here.

3 minutes stretchbreak

1. Rub your hands until they get warm;
2. Blow some air into your fist;
3. Pat your arms and your shoulders;
4. Jump up and down three times;
5. Squat three times;
6. Repeat this one more.

Finally, we can take a flight home to solid ground, where the climate is more forgiving.

Coming home, there is still some housekeeping and gardening to do.

Under the Rug

Welcome home. Time to do some thorough housekeeping. Scrum is all about exposing the relative efficacy of current management, environment, and work techniques so that improvements can be made. That means we must look closely at what the team and broader organization are sweeping under the rug.

This play provides the opportunity to reveal some of the unaddressed challenges. It encourages participants to be open about challenges. Those can be (but are not limited to):

- Things are unaddressed due to a lack of ownership and accountability. "Oh, that sandwich rotting away in the fridge... not mine!"
- The roadmap doesn't allow time to address them. "No, we can't! We're already falling behind schedule."
- Developers don't have a sense of professionalism that what they create ultimately represents them. "Yes, I know it's messy, but it's a temporary workaround. I'll fix it later when I have time." (which is never)
- Lack of empathy and understanding of the users. "It's not a bug; it's a feature!"
- The work is dumped, not delivered. "They're using it wrong!"
- Not being outcome-driven. "I know it's crap, but I did my thing right."
- Hiding in shadows and silos: "I'm a programmer; I'm not responsible for testing and quality."
- Being conformist: "It's always been done this way."

7,5 minutes

Provide each participant with a *mini rug* and a set of sticky notes. Participants may be invited to answer the following questions on individual sticky notes. They may then place them under their rug.

- What is giving us speed now but will end up slowing us down?
- What's bugging users that we are not addressing?
- What did I do that I am not proud of?
- What problems might disappear if users had a better understanding of the possibilities?
- What are things we don't like doing that we do because it's always done that way?
- What is not adding any value yet slowing us down?

10 minutes

Pair up. Reveal both rugs. Consolidate what's under both rugs and remove duplicates. Merge pairs. Consolidate. Everyone together, consolidate and review the final result.

Good Housekeeping

It's safe to say that your mother doesn't clean your codebase.

Good Housekeeping: "Maintain a completely clean product and work environment continuously."

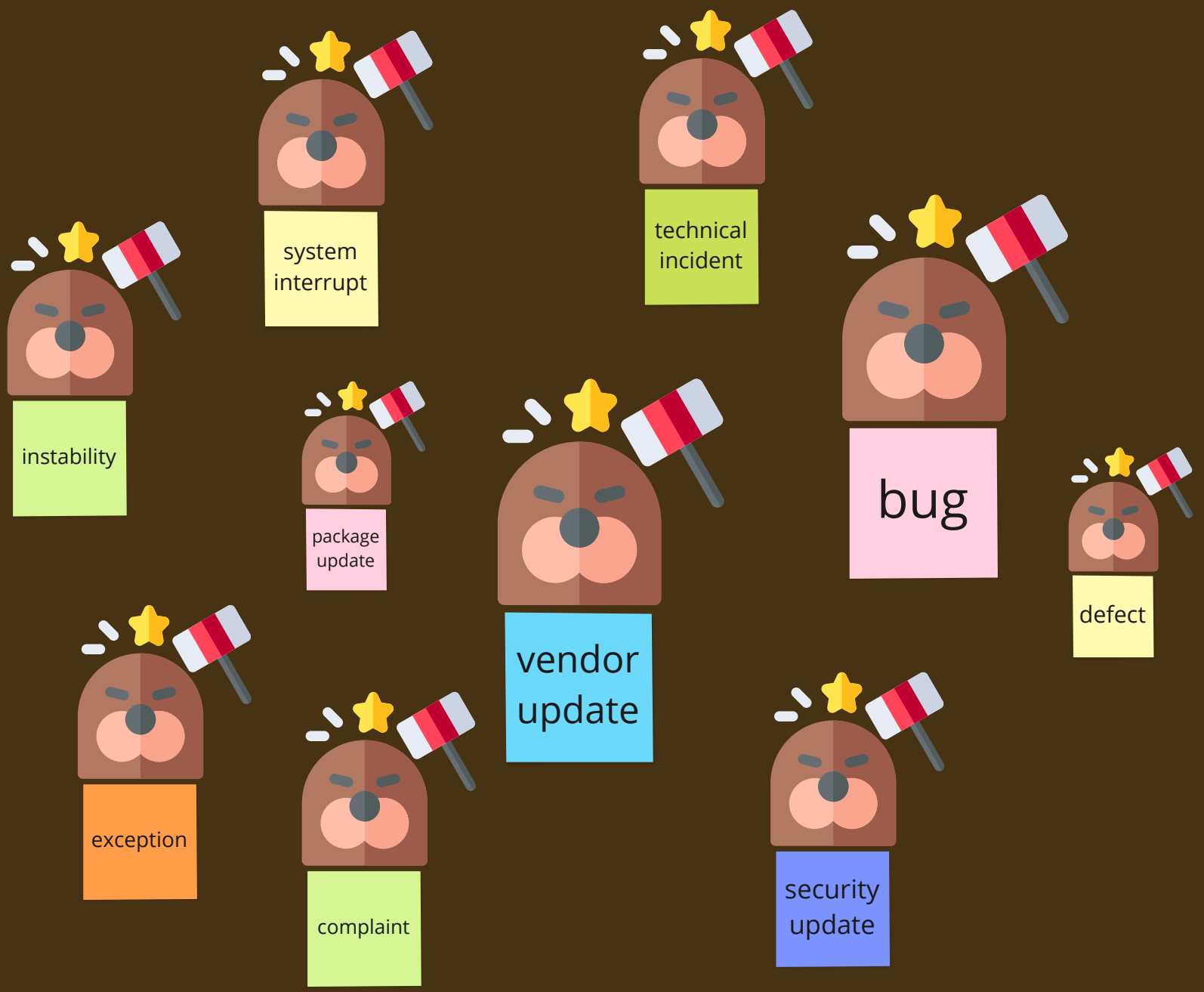
Good Housekeeping may refer to good practices in maintaining (for example) the product, the work environment, tooling, code, design, and architecture. What practices and when to use them is up to the Developers. They are accountable for Good Housekeeping. How they go about it is their business.

Here are some examples of practices. Can the crew determine if they are considered "good" or "bad" Housekeeping?

- DRY: Do not repeat yourself is generally considered "good" Housekeeping.
- Patch-fix: Only addressing the symptom is generally considered "bad" Housekeeping, as problems will continue to emerge.
- Hack-fixes: This is generally considered "bad" Housekeeping. It contributes to Obscurity and Fragility.
- Crystal Balls: Add presently unneeded patterns, protocols, and conventions because you *might* need them in the future. This is considered "bad" Housekeeping. It obscures.
- Boy Scout Rule: Always refactor the code/design in a better state than how you found it. This is generally considered "good" Housekeeping.
- Quick-fix: Fix it first, refactor when there is time. This is generally considered "bad" Housekeeping as developers rarely return to it.
- Wax on, Wax off routine: For adding anything new, refactor something existing. This is generally considered "good" Housekeeping.
- Keep out: If it ain't broken, don't touch it. This is generally considered "bad" Housekeeping. Although it may not be broken it may be obscure, rigid, fragile, or immobile.
- Wait until it breaks: only install updates when there is an absolute need. This is considered "bad" Housekeeping. Preventing is better than repairing.
- Wack-a-Mole: fix defects as soon as they emerge. No buglists! This is generally considered "good" Housekeeping.

15 minutes In Quads, list and discuss examples of "good" versus "bad" housekeeping.

Attribution: Good Housekeeping is a Scrum Pattern by Neil Harrison.


system interrupt
technical incident
instability
package update
vendor update
bug
defect
exception
complaint
security update

Whack-a-Mole

“Whack-A-Mole” is an arcade and carnival game, known initially as Mogura Taiji (“Mole Buster”) in Japan.

“Immediately resolve product problems, big and small, as they arise.” – James O. Coplien.

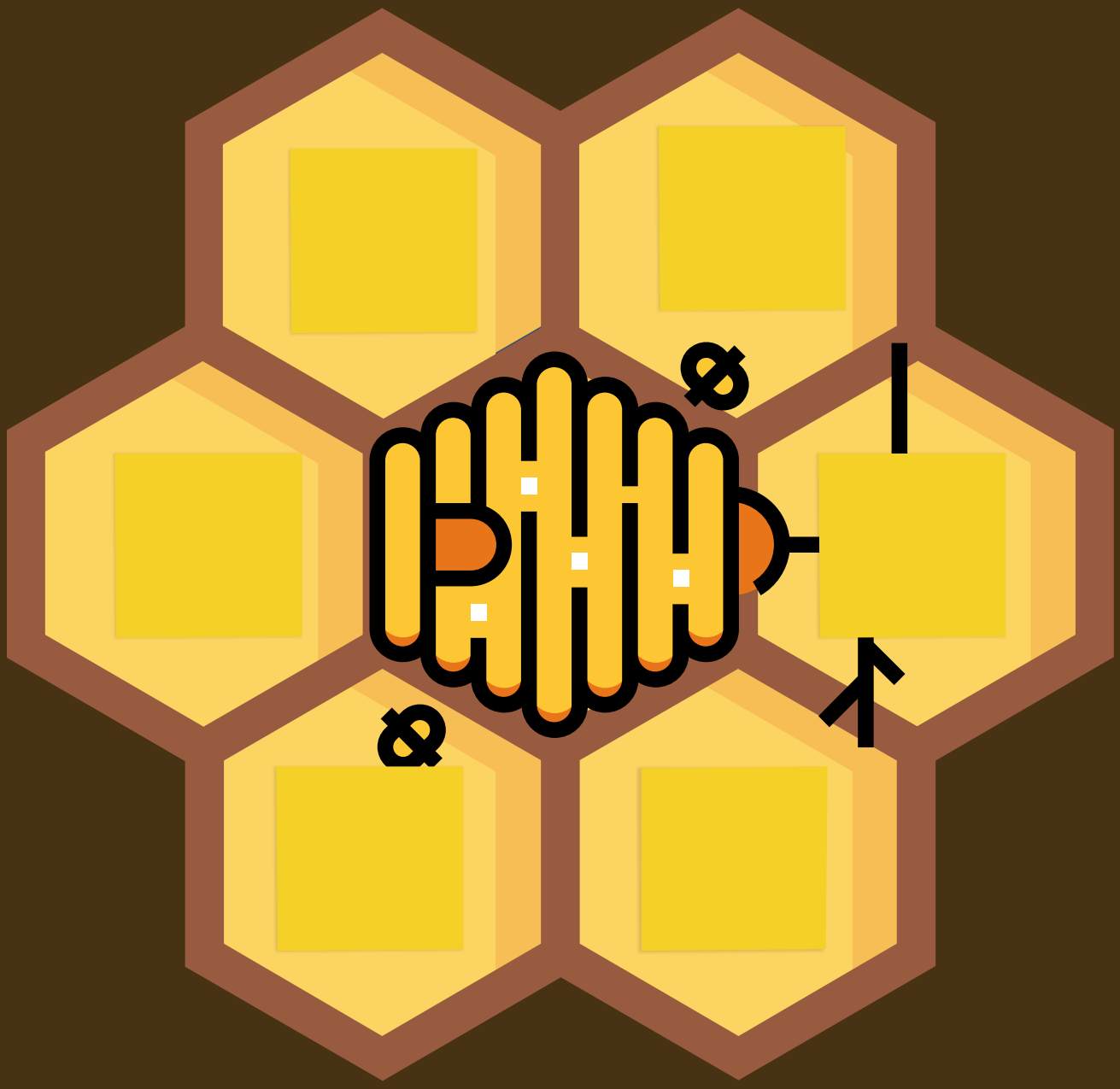
Rather than keeping a buglist or adding defects to a backlog, address them directly. The team should prioritize fixing a broken, disrupted, interrupted, unstable or faulty product, or a product that does the wrong thing, over enriching the product with new capabilities. After all, the presence of any issue means some aspect of product value is disrupted. What is previously held to be “done” was not “done”!

Scrum Developers learn that it’s more effective to stop moles from entering the system in the first place. They can do this through built-in quality and Good Housekeeping. A Scrum Team should ensure that what is working, remains working.

Scrum Developers are much like gardeners. Don’t allow moles to mess up your beautiful garden. Deal with moles as they emerge.

2 minutes	The gardeners individually write down a list of unexpected disruptions (moles) that frequently emerge during development.
3 minutes	Pair up. Merge this list with your partner (remove duplicate moles).
4 minutes	Merge two (or three) pairs. Merge the lists (remove duplicate moles).
6 minutes	With the whole group, merge all lists to create one master list (remove duplicate moles).

This play is based on the similarly named Scrum Pattern by James O. Coplien.



Money on the Honey

"Well," said Pooh, "what I like best..." and then he had to stop and think. Because although 'Eating Honey' was a very good thing to do, there was a moment just before you began to eat it which was better than when you were, but he didn't know what it was called. – A. A. Milne, Winnie-the-Pooh

"Money on the Honey" is a simple play that literally makes learning stick. (Money = valuable learning, Honey = sticky note). Participants can create their own 1-3 word cheat sheets (things they would like to remember from the training). They can write down a maximum of six important learnings on sticky notes. A sticky note may only contain a maximum of three words. The participants may then stick these notes on highly visible locations. For example, they can stick it on: each other, or a coffee machine, mug, monitor, laptop, mobile phone, clock window, or door. This may trigger conversations as colleagues wonder and ask about the sticky notes.

Let's try this with a pattern called Swarming. After all, bees operate in swarms and make honey. Swarming is to focus maximum team effort on a single shared objective, for example, a Sprint Goal, team improvement, impediment, or Product Backlog item. A captain leads the swarm. The Captain is in possession of a specific object (for example, a stuffed animal). Everyone on the team must help the Captain. No one can interrupt the Captain.

3 minutes	Individually, participants brainstorm a maximum of 6 benefits of swarming.
4 minutes	They may now reduce this list to keywords only. They may be written down on sticky notes. Only three words are allowed per sticky note.
6 minutes	The participants may place the sticky notes around their (home-) office.

Some of the positive effects of Swarming can be: Decreased lead time (item age), enhanced collective intelligence, removal of the shade, transparency, reduced individual dependencies, collective ownership, and instilling quality.

Attribution: Swarming is a Scrum Pattern outlined by Jeff Sutherland

deepdiving

What surprised you about deepdiving?

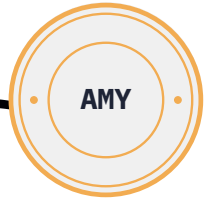
What doubts and questions do you still have?

What would like to try in your practice?

What sounded unrealistic to you?

What sounded like common sense?

What sounded like **uncommon** sense?




R2M
Deep Diving
- Developer Culture -

Appendix: 4C Map

The R2M-VE applies a 4C baseline from Training From the Back of the Room, Virtual Edition (TBR-VE) by Sharon Bowman.

1. Culture & Ego

Learning Outcomes:

1. Teaching-back professional developer practices participants are already familiar or proficient in;
2. Identifying what mindset drives what type of *error culture*;
3. List what is made possible when the Scrum Values are lived;
4. Identify various ego-types and their pitfalls;
5. Listing ways a *hero* can overshadow and limit the growth of other team members.

Plays:

1. Mad Tea Party (C1)
2. Error Culture (C2)
3. Deep Dive (C3)
4. Yellow Submarine (C2)
5. Remove the Shade (C3)
6. Stretch Break (C4)
7. Checkpoint 1 (C4)

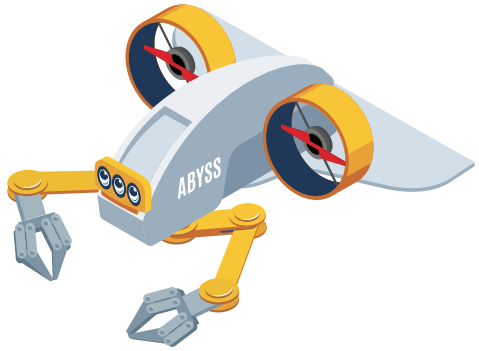
2. Technical Debt

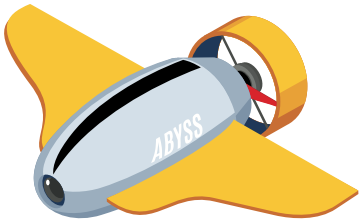
Learning Outcomes:

1. Identifying the four horsemen of Technical Debt;
2. Defining essential characteristics of a Bug Report;
3. Defining good patterns for Issue Analysis;
4. Defining good patterns for Quality Assurance;
5. Creating a pattern for responding to unexpected challenges;
6. Record a Captains Log for essential observations and learnings;
7. Co-create a Product Telemetry Radar;
8. Identify "leading" and "lagging" indicators;
9. Practicing useful patterns for identifying, visualizing and resolving Technical Debt.

Plays:

1. Four Seahorsemen (C1)
2. Something's Fishy (C2)
3. Unexpected Encounters (C3/C4)
4. Captains Log (C3/C4)
5. Radar (C2/C3)
6. Ghost Trapper (C3)
7. Stretchbreak (C4)
8. Checkpoint 2 (C4)





3. Releasing/Deploying

Learning Outcomes:

1. Identifying and connecting various patterns for testing;
2. Visualizing version control strategies;
3. Listing good, lightweight patterns for deploying safely to production.

Plays:

1. Connect the two (C1)
2. Branching and Merging (C2/C3)
3. Breaking Through (C3)
4. Stretchbreak (C4)
5. Checkpoint 3 (C4)

4. Good Patterns

Learning Outcomes:

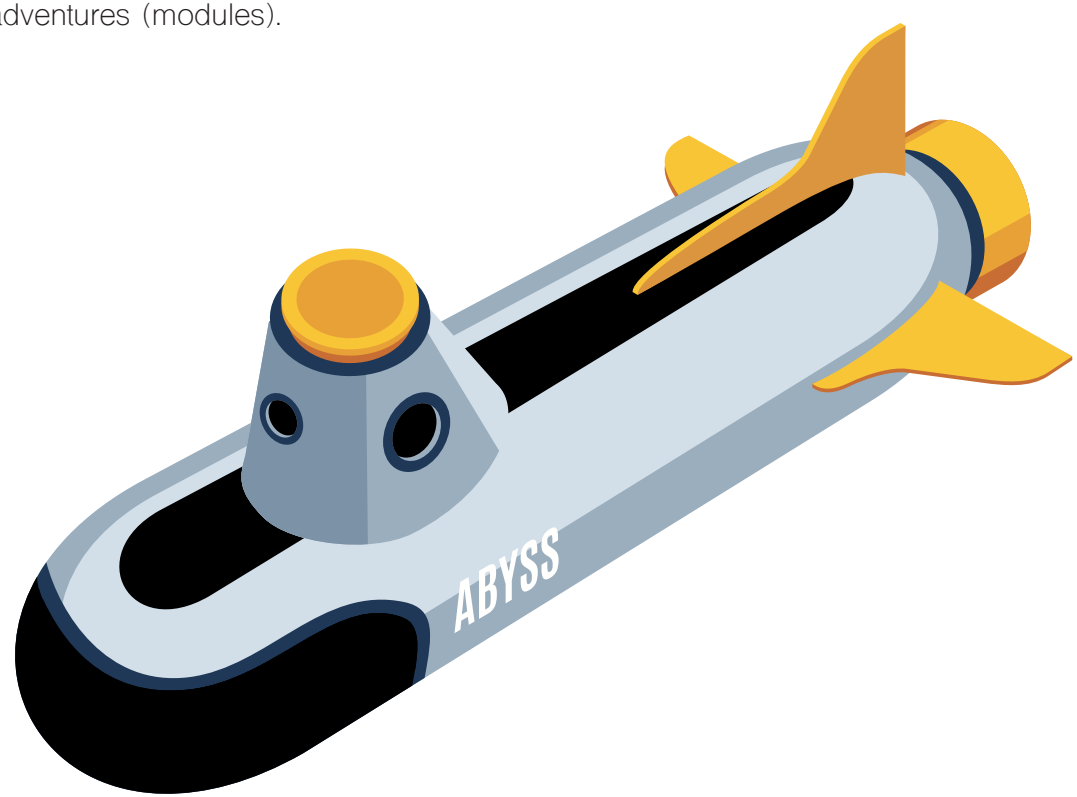
1. Identifying bad patterns;
2. List examples for good and bad patterns for maintaining a (clean) product and work environment;
3. List examples of unexpected and disruptive disturbances/events;
4. Brainstorm benefits of Swarming
5. Identifying obstacles that get in the way of professional patterns.
6. Reflecting on individual doubts and concerns when it come following professional patterns.

Plays:

1. Under the Rug (C1)
2. Good Housekeeping (C2)
3. Wack a Mole (C2/C3)
4. Money on the Honey (C2/C3)
5. Traveljournal (C4)

The Road to Mastery (R2M) is a Scrum Learning Journey containing 12 adventures (modules).

1. Basecamp
2. Agile Backpacking
3. The Game of Scrum
4. Living the Scrum Values
5. Mountaineering Scrum together
6. Kayaking the Value Stream
7. Surviving Self-Management
8. Deep Diving Developer Culture
9. Exploring Artifacts
10. Smooth Sailing the Events
11. Bootcamp
12. Coaching Dojo



Acknowledgement

The Road to Mastery (R2M) is developed by Sjoerd Nijland.
The R2M is published at Serious Scrum; AGNC.

It builds on and is inspired by on the works of:
- Ken Schwaber and Jeff Sutherland: the Scrum Guide;
- Sharon Bowman: Training from the Back of the Room (TBR);
- Evelien Roos: Training from the Back of the Room Virtual Edition (TBR-VE);
- Project Zero: a research center at the Harvard Graduate School of Education;
- Henri Lipmanowicz and Keith McCandless: Liberating Structures;
- Scrum Patterns by Jeff Sutherland, James Coplien e.o.;
- Freepik for vector illustrations;
- And others attributed in the various plays.

For feedback and questions, please contact:
sjoerd.nyland@gmail.com

Comment on attribution:
“I aim to diligently attribute anyone who deserves credit or reference and refer to those sources of inspiration from which the Road to Mastery emerged. Please reach out with suggestions and comments on where this can and should be improved.” - Sjoerd Nijland.

Serious Scrum R2M

Deepdiving Developers

Sjoerd Nijland

www.road2mastery.com