

# What is the first level of Continuous Integration?

Build code on a regular basis	Release code on a regular basis to production	Merge everyone's code on a regular basis	Make a potential product (releasable artifact) on a regular basis	Test code on a regular basis

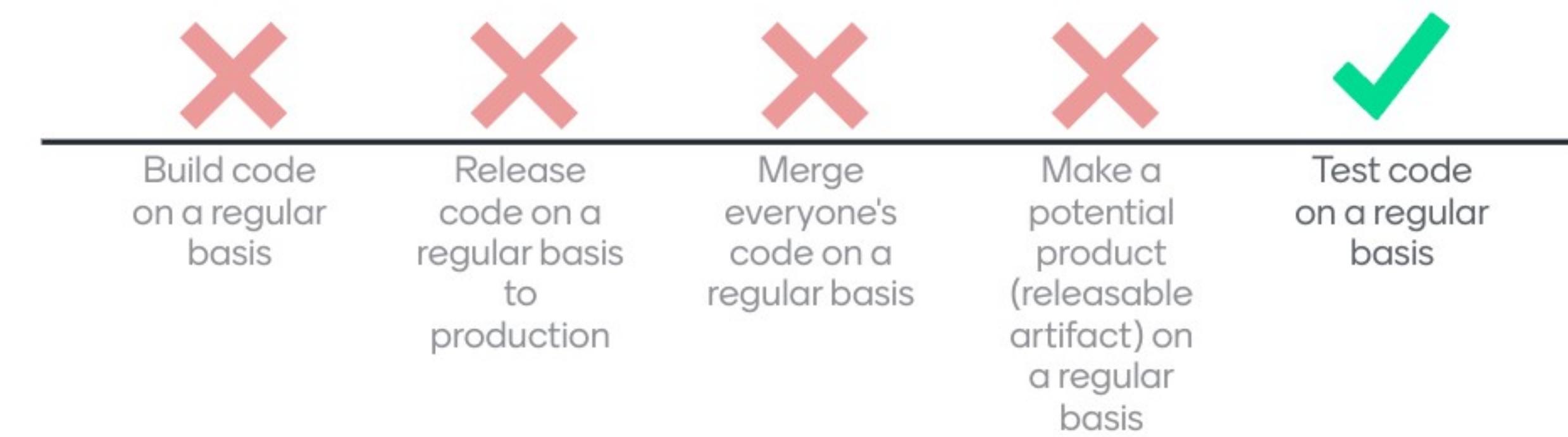


# What is the second level of Continuous Integration?

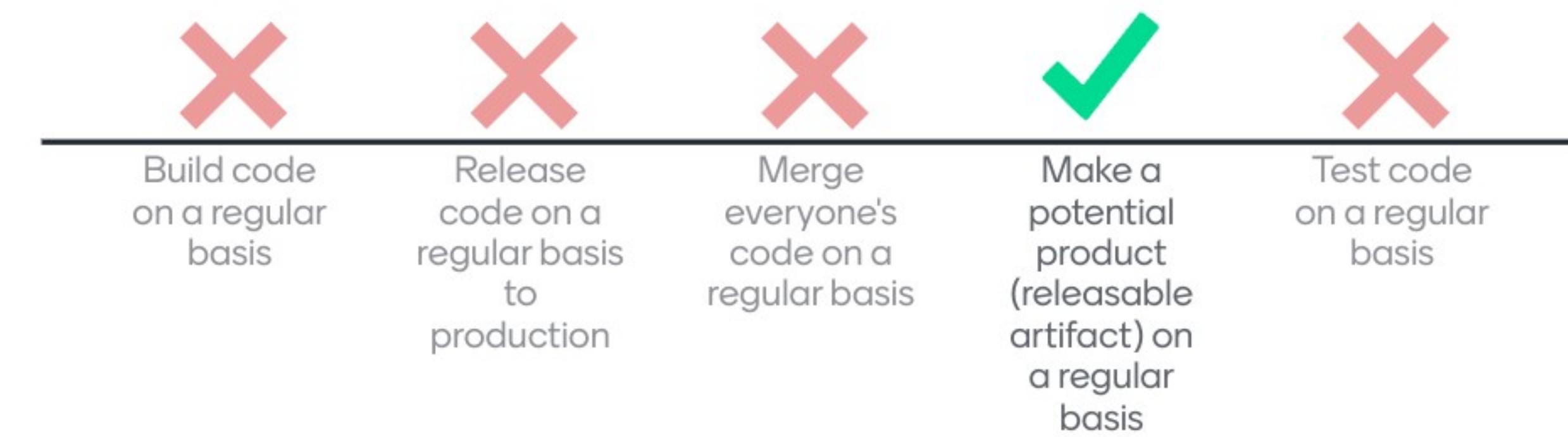
Build code on a regular basis	Release code on a regular basis to production	Merge everyone's code on a regular basis	Make a potential product (releasable artifact) on a regular basis	Test code on a regular basis



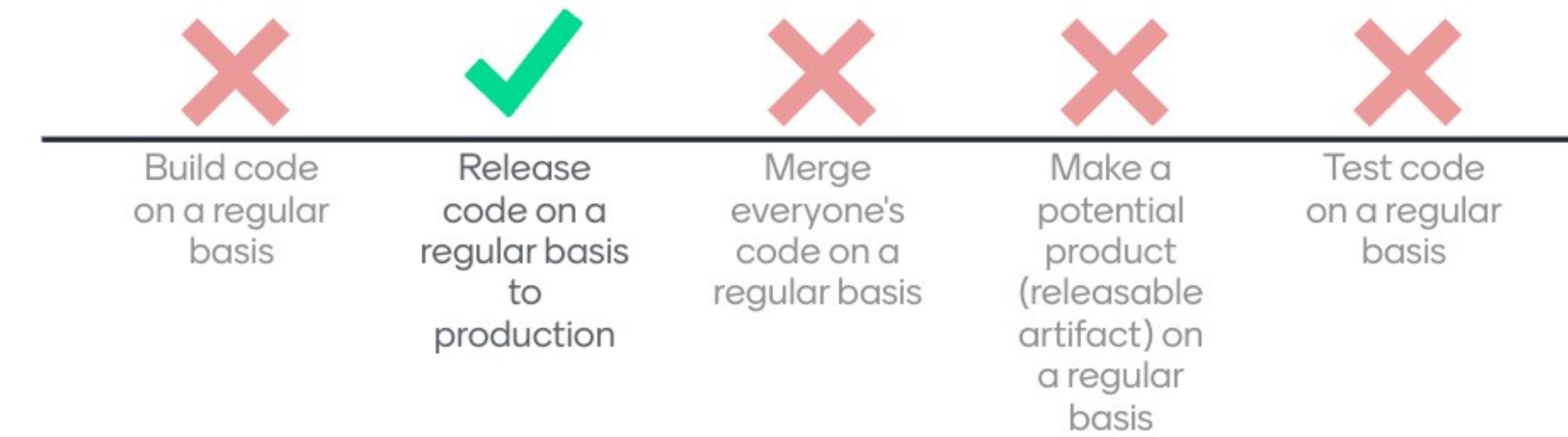
# What is the third level of Continuous Integration?



# What is the fourth level of Continuous Integration (Continuous Delivery)?



# What is the last level of Continuous Integration (Continuous Deployment)?



# Responding quickly to changing requirements?



# Releasing working products often?



# Sustainable pace?



# Standup meeting



# Scrum planning meeting



# Scrum review meeting



# Scrum retrospective meeting



Game must render at 60fps



must be able to mark a book "read"



---

Functional  
Requirement



Non-functional  
Requirement

must be able to mark a book "read"



# API is documented using JavaDoc



User must be 18 years old to purchase



The software product must come in three tiers: home, professional, and enterprise



# Customization?



# Guiding user through steps (wizards)?



Users are more likely to ...



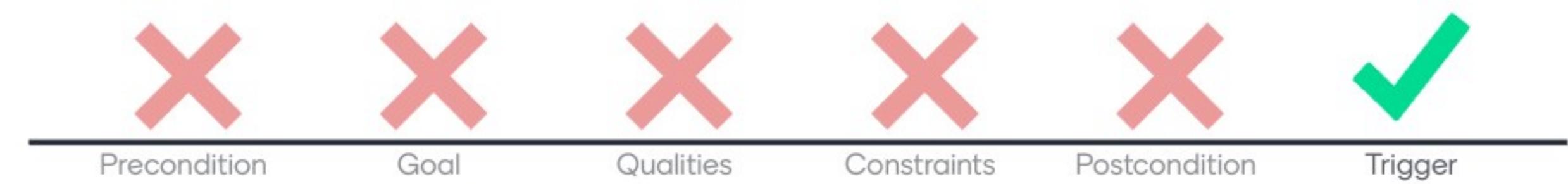
A user story with that can (and should) be split up into many smaller user stories...



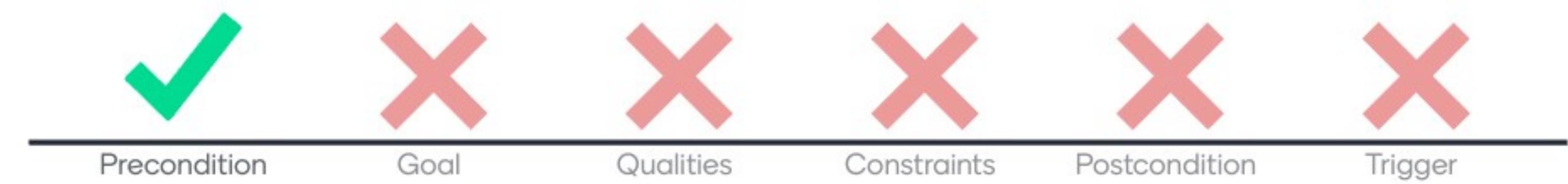
# We should AVOID putting what in user stories...



The first thing the actor does =



# The things the actor knows & has



Usually has some non-functional requirements (NFRs)



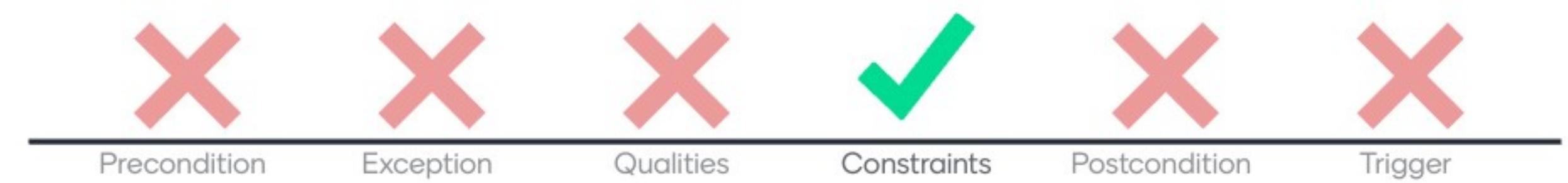
# What the actor is trying to do



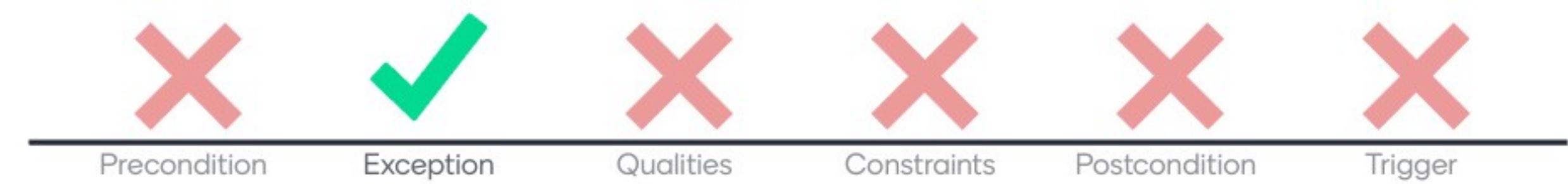
# What the actor gets out of it if goal is met



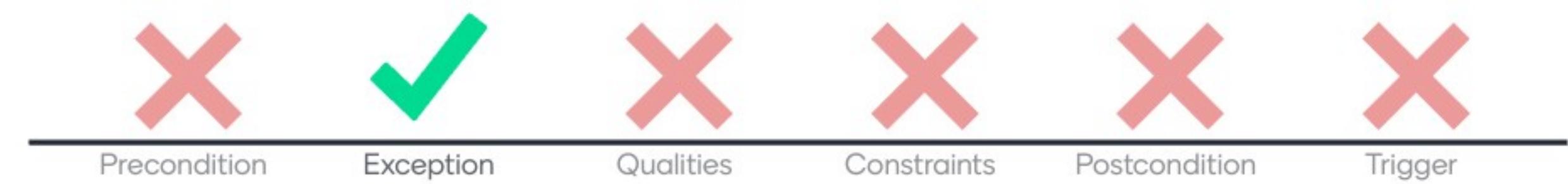
# Restrictions on technical workings



Uses the same numberings as the basic flow



Uses the same numberings as the basic flow



User stories should be...



User stories should be...



# User stories should be...

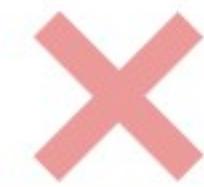


Able to be  
implemented with  
existing software  
& hardware



Motivating the  
creation of new  
software &  
hardware

# User stories should be...



For anyone to  
ensure a broad  
appeal



For the people  
we're making the  
software for

# User stories should be...



Inspirational big  
steps, "vision"



Small steps, can  
be implemented in a  
short amount of  
time

# User stories should be...



# User stories should be...



Final, not  
changing, so  
development isn't  
interrupted



Changable, we  
can adjust what  
they are and  
mean

# User stories should be...



# User stories should be...



# User stories should be...



Finished when the  
acceptance tests  
all pass

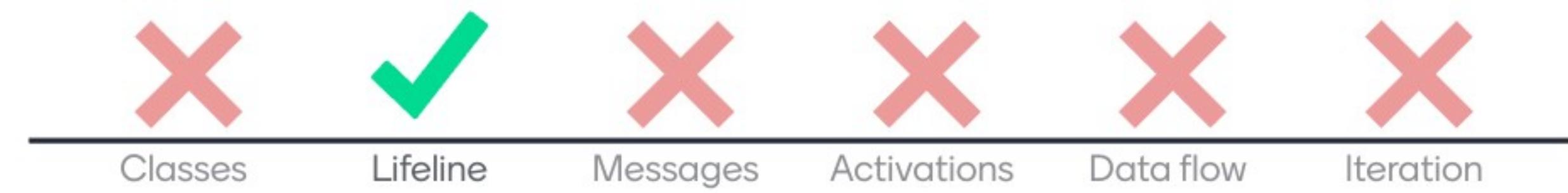


Finished when  
developers & users  
agree the software  
fulfills them

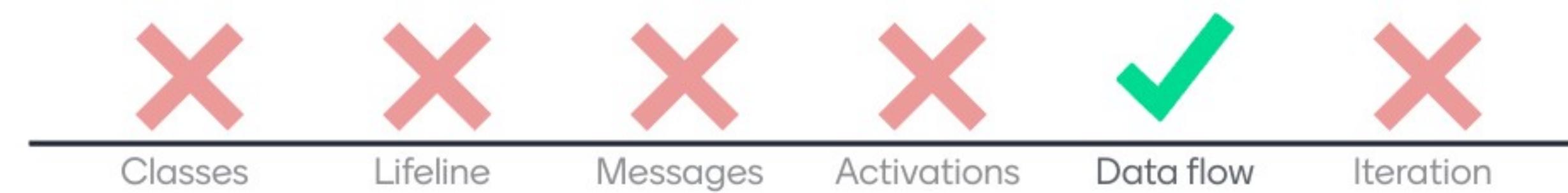
In a UML Sequence Diagram, time goes...



In a UML sequence diagram, vertical dashed lines...



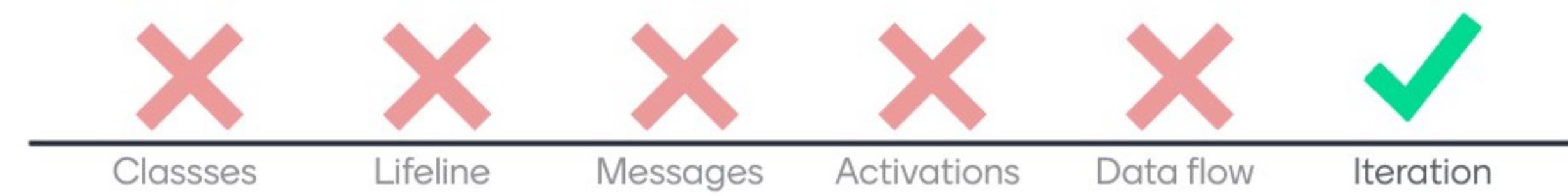
In a UML sequence diagram, horizontal dashed lines...



In a UML sequence diagram, columns represent...



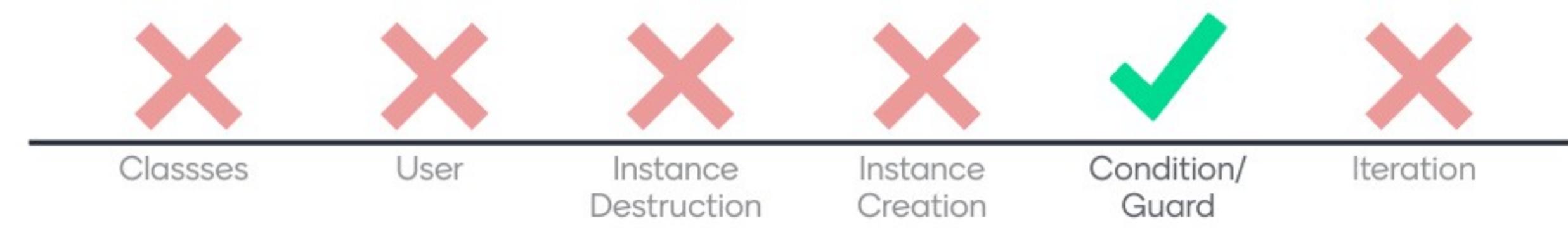
In a UML sequence diagram, \* represents...



In a UML sequence an arrow pointing to the top of a column represents...



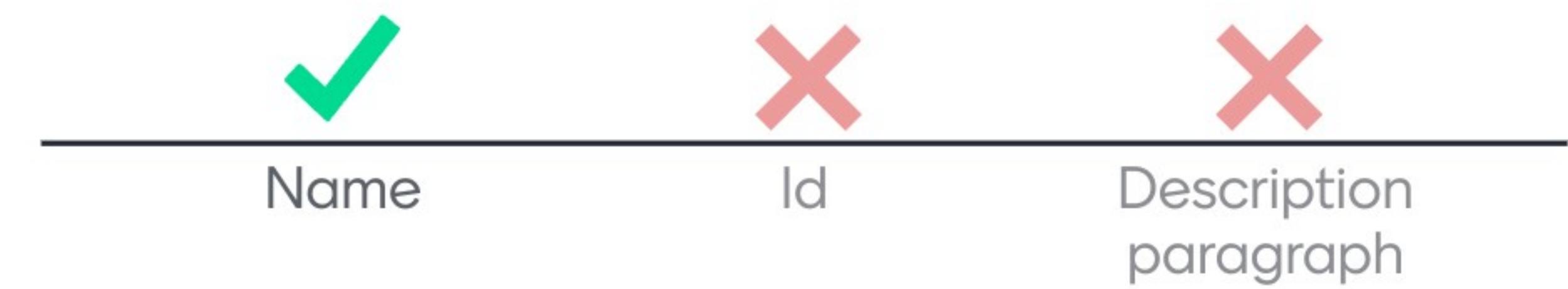
In a UML sequence square brackets [] represents...



In a UML sequence stick figure represents...



# 1st thing in a Use Case...



# Next thing in a Use Case...



# Next thing in a Use Case...



# Next thing in a Use Case...



# Next thing in a Use Case...



# Next thing in a Use Case...



# Next thing in a Use Case...



Basic Flow



Link to a sequence  
diagram

# Next thing in a Use Case...



# Next thing in a Use Case...



# Next thing in a Use Case...



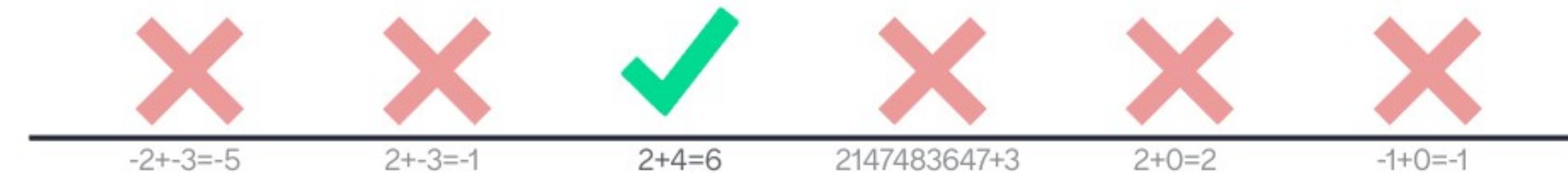
# Black box testing



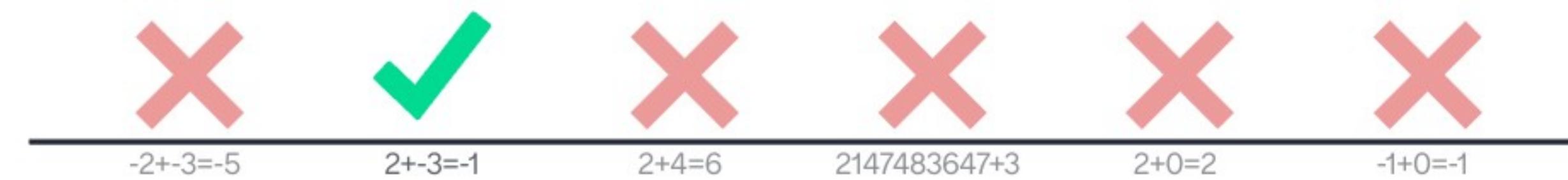
# Regression testing

	Write tests about the faults we fixed		Write tests about our automated systems		Write tests about data structures being used
---	---------------------------------------	---	---	---	--

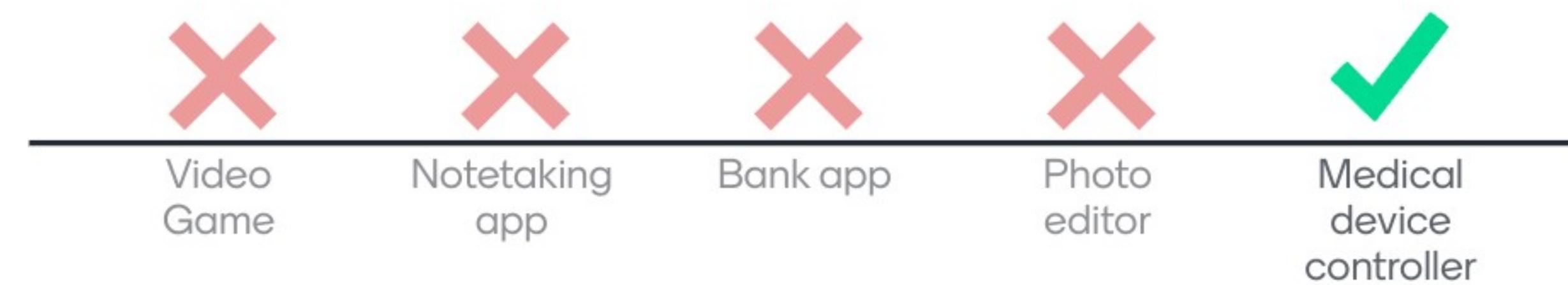
Equivalence classes: same class as  $2+3=5$  for a function `add(int, int)`



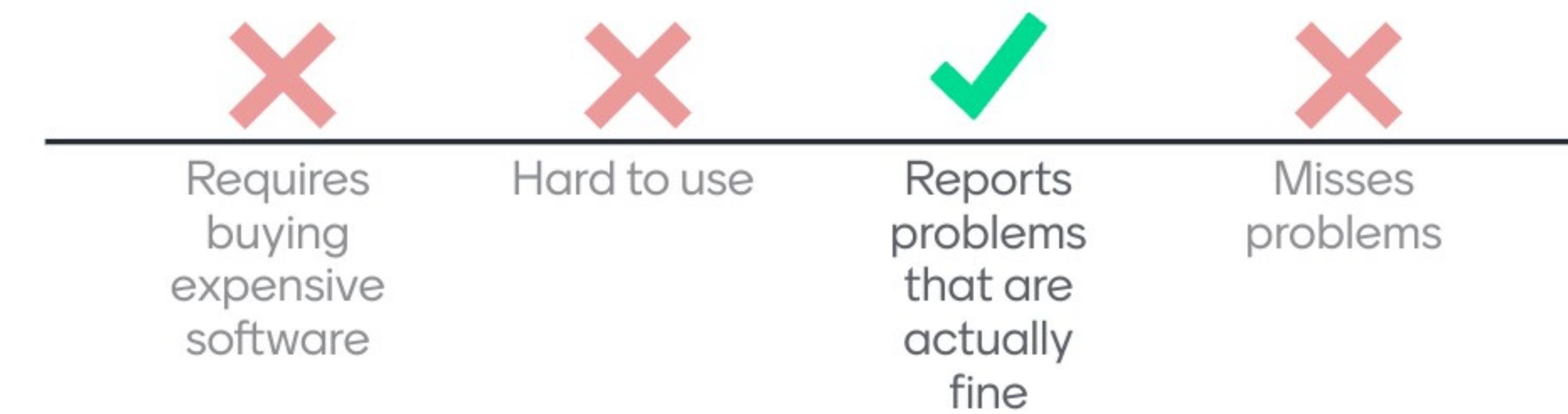
Equivalence classes: same class as  $57+(-27)=30$  for a function `add(int, int)`



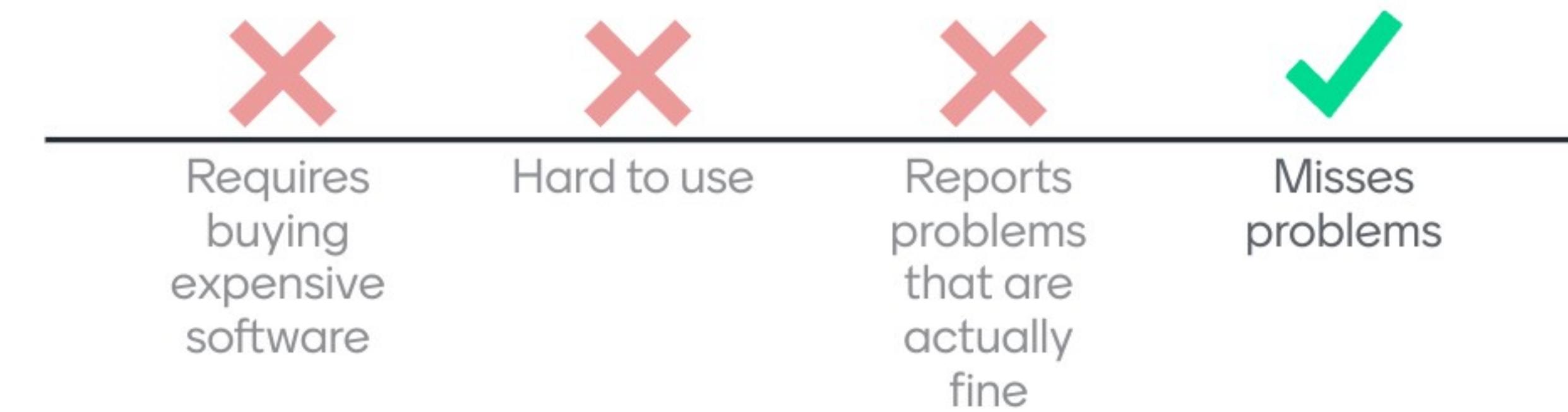
# Which is likely to have the highest testing requirements?



# What's the main problem with static analysis?



# What's the main problem with test functions (JUnit)?



# Black box testing

		
Write tests about the code we're testing	Write tests about the interfaces we're testing	Write tests about the the NFRs we're testing

# Regression testing

	Write tests about the failures we fixed		Write tests about our automated systems		Write tests about data structures being used
---	---	---	---	---	--

# Top-down testing



Write tests for  
the classes  
with few  
imports first



Write tests for  
the classes  
with some  
imports first



Write tests for  
the classes  
with many  
imports first

Code that is hard to write tests for is also hard to...



add features to



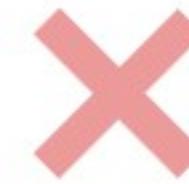
read and  
understand what  
it does

# Writing tests at the end of a project is bad because...

it's hard to determine what features are broken	it's hard to determine when features got broken	it's hard to determine where features are broken	you tend to run out of time and have to skip writing tests



# What to developers often forget to test



successful  
execution (no  
exceptions)



exceptional  
execution (some  
error or  
something)

# Testing a single class or method...



# Testing multiple classes/methods working together...



Testing large chunks of the software from one end to the other...



# What do we do with small bits of code that are repeated exactly?



Replace them  
with a simpler  
or more  
expressive  
alternative

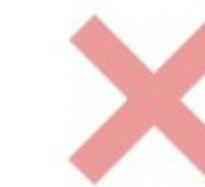


Extract them  
to a function/  
method/  
class/library



Recognize  
them as an  
idiom or  
pattern

What do we do with small bits of code that are repeated but always a little too different to extract to a common library?



Replace them  
with a simpler  
or more  
expressive  
alternative



Extract them  
to a function/  
method/  
class/library



Recognize  
them as an  
idiom or  
pattern

It's important to write idiomatic code to



# Design patterns let us



One class has a list of objects that it calls a method on every time something changes



Class that can only be instantiated once and there's only one way to access that instance



Each instance has a list of instances with the same superclass arranged in a tree



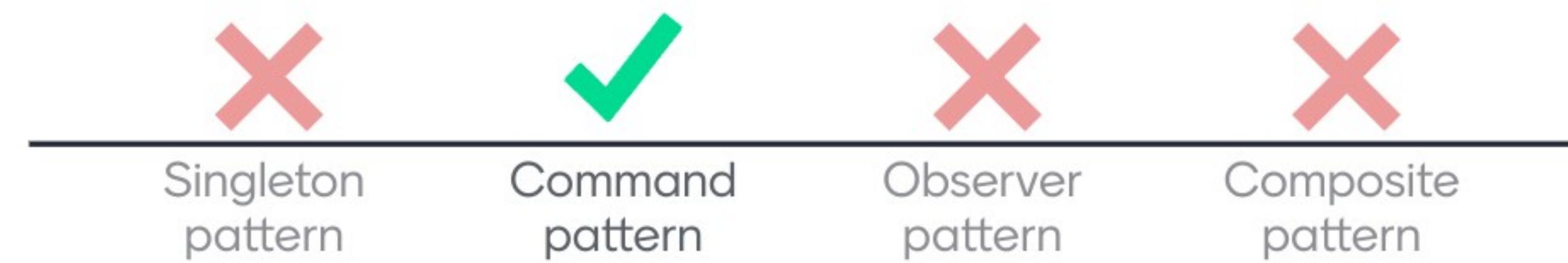
One class has a list of objects that it calls a method on every time something changes



Class that can only be instantiated once and there's only one way to access that instance



# Class with instances that represent actions or changes



Each instance has a list of instances with the same superclass arranged in a tree



The template method for the template method pattern goes in the



The abstract methods for the template method pattern go in the



The override methods for the template method pattern go in the



"hooks" are similar to

