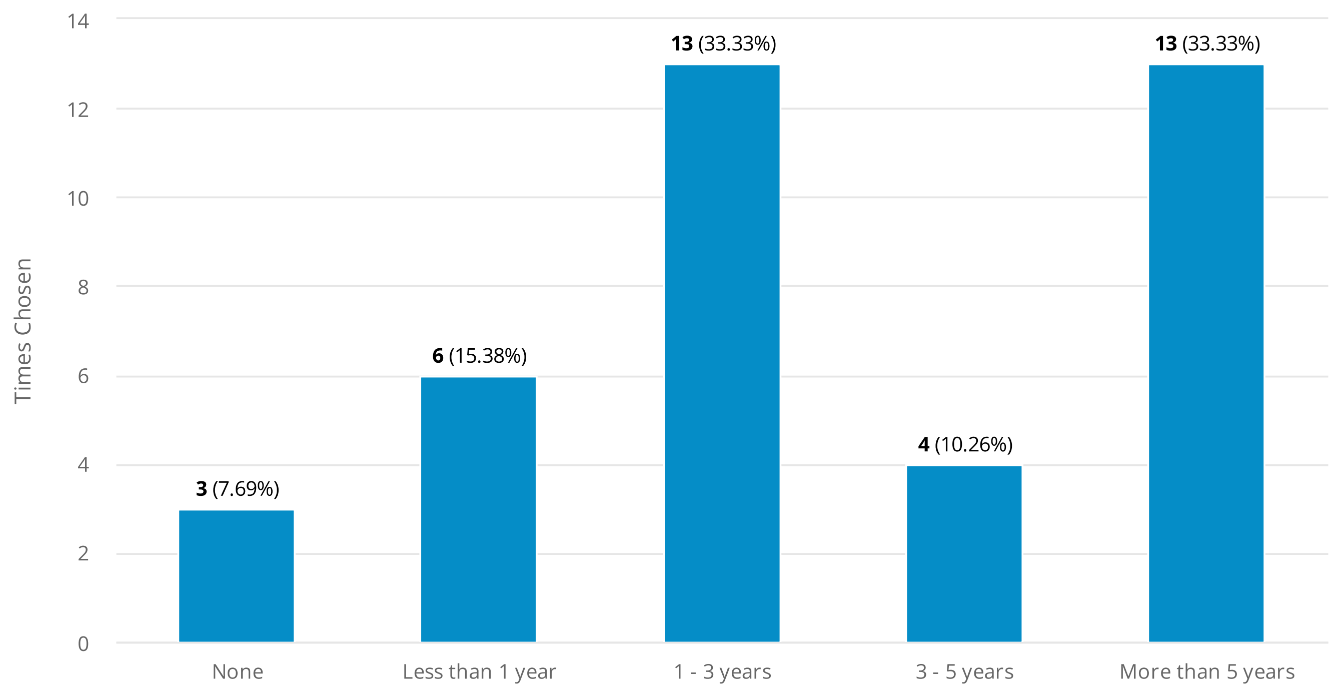
Design Patterns in Game Development

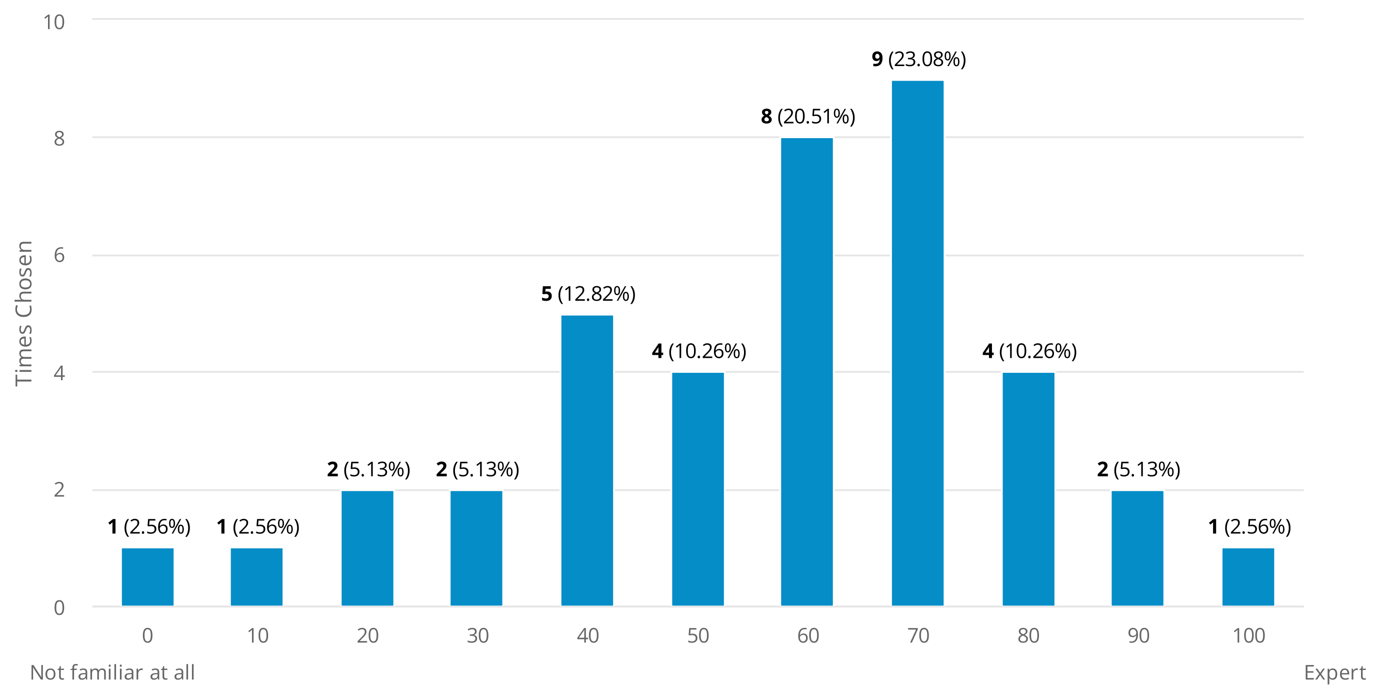
How many years of experience do you have in software development / programming?

Number of responses: 39



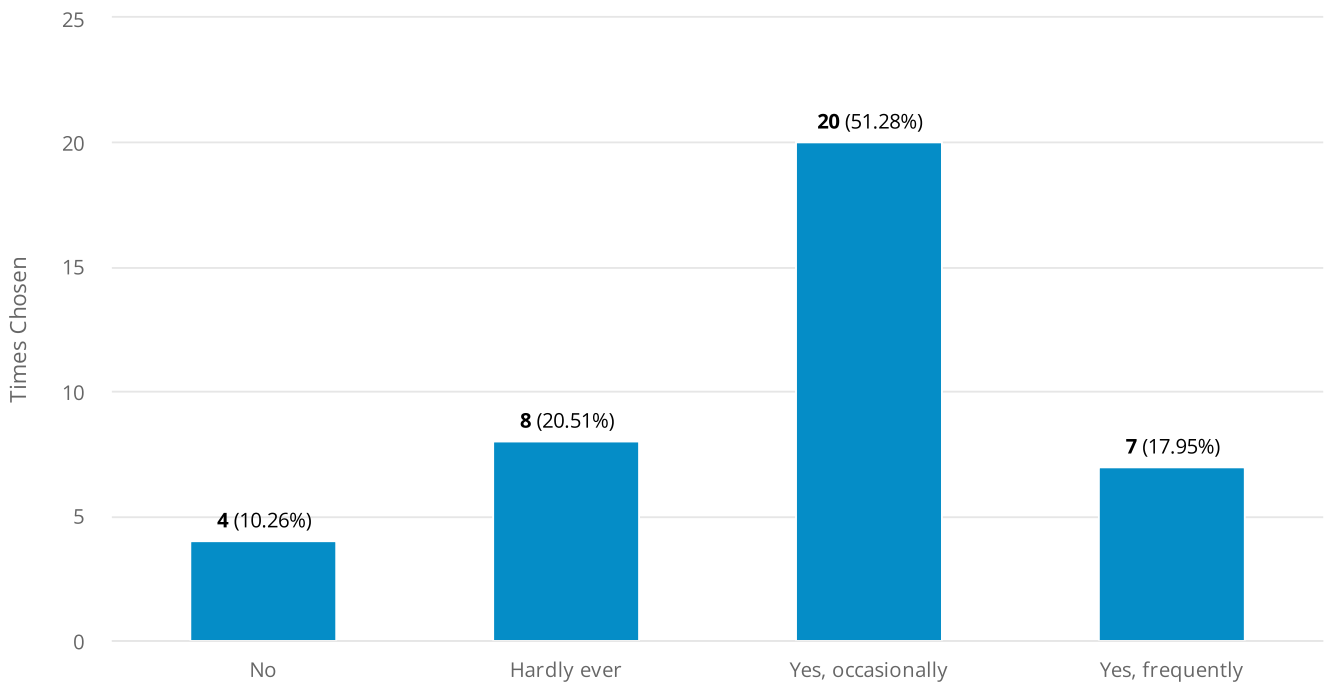
How familiar are you with design patterns in software development?

Number of responses: 39



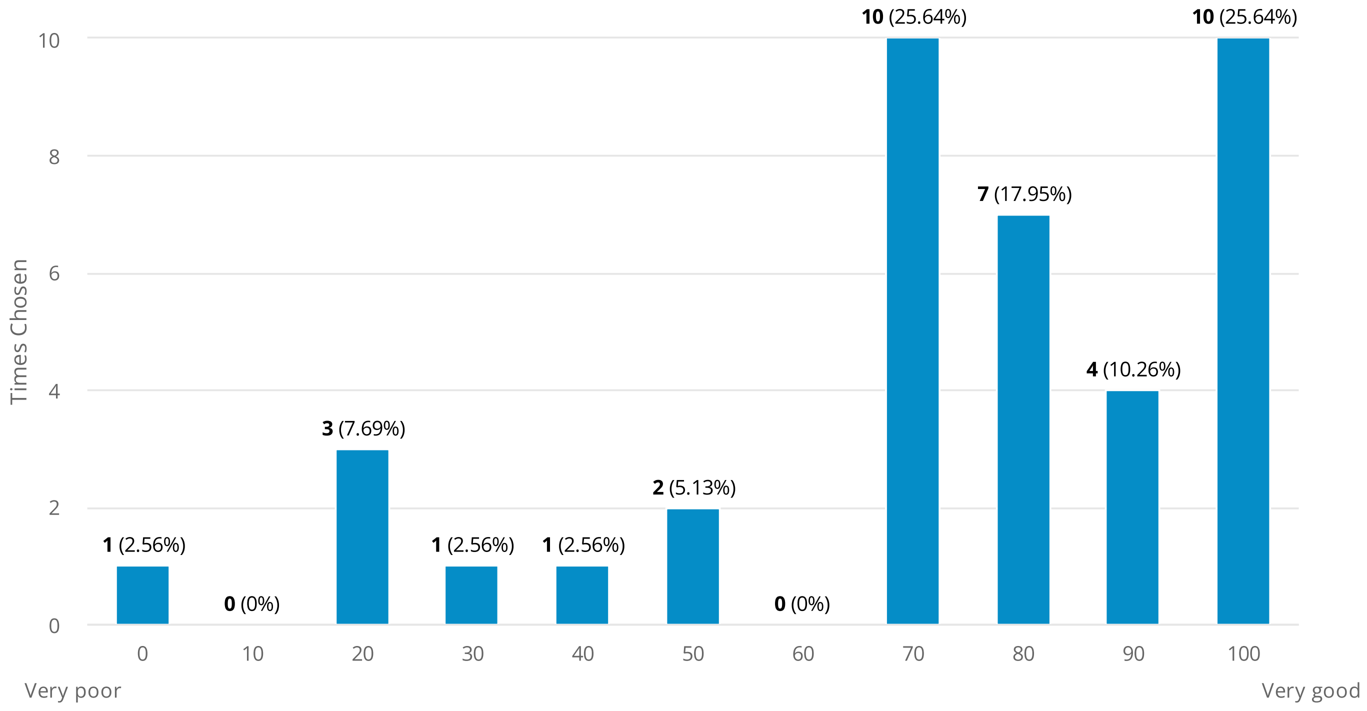
Have you ever used design patterns in one of your projects?

Number of responses: 39



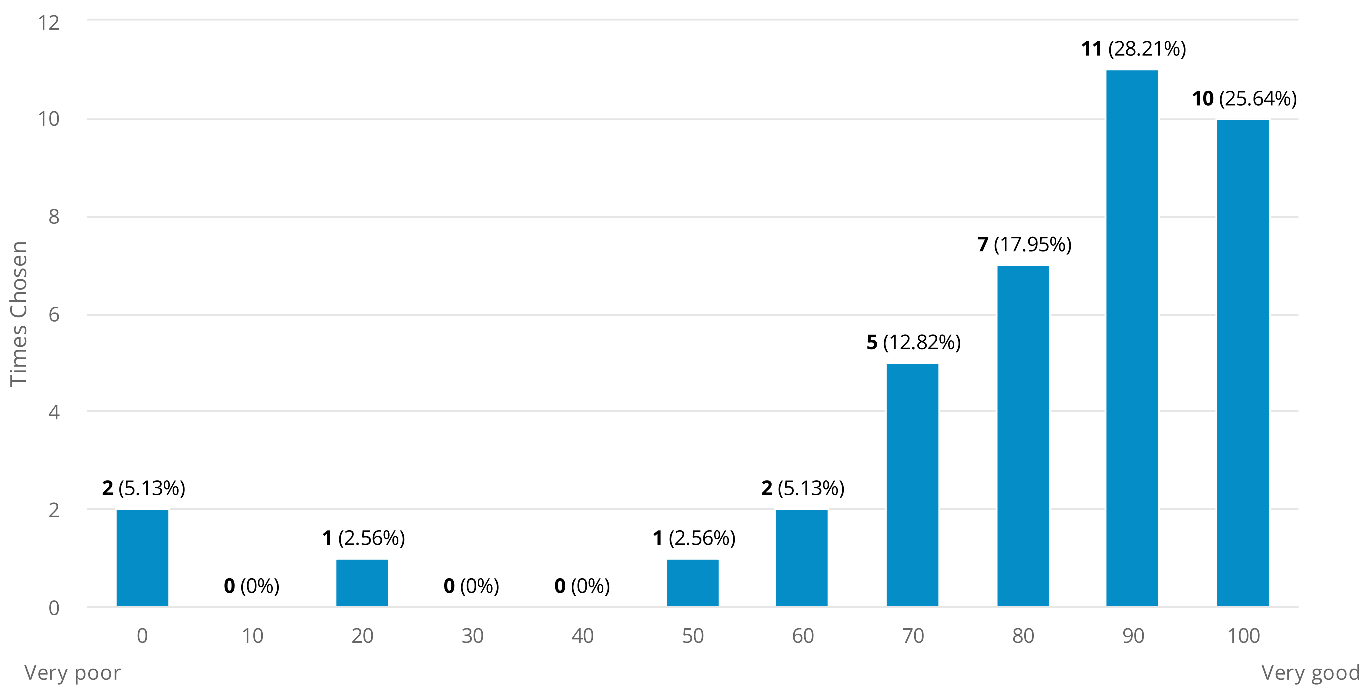
How would you rate the readability of the code in the prototype with design patterns?

Number of responses: 39



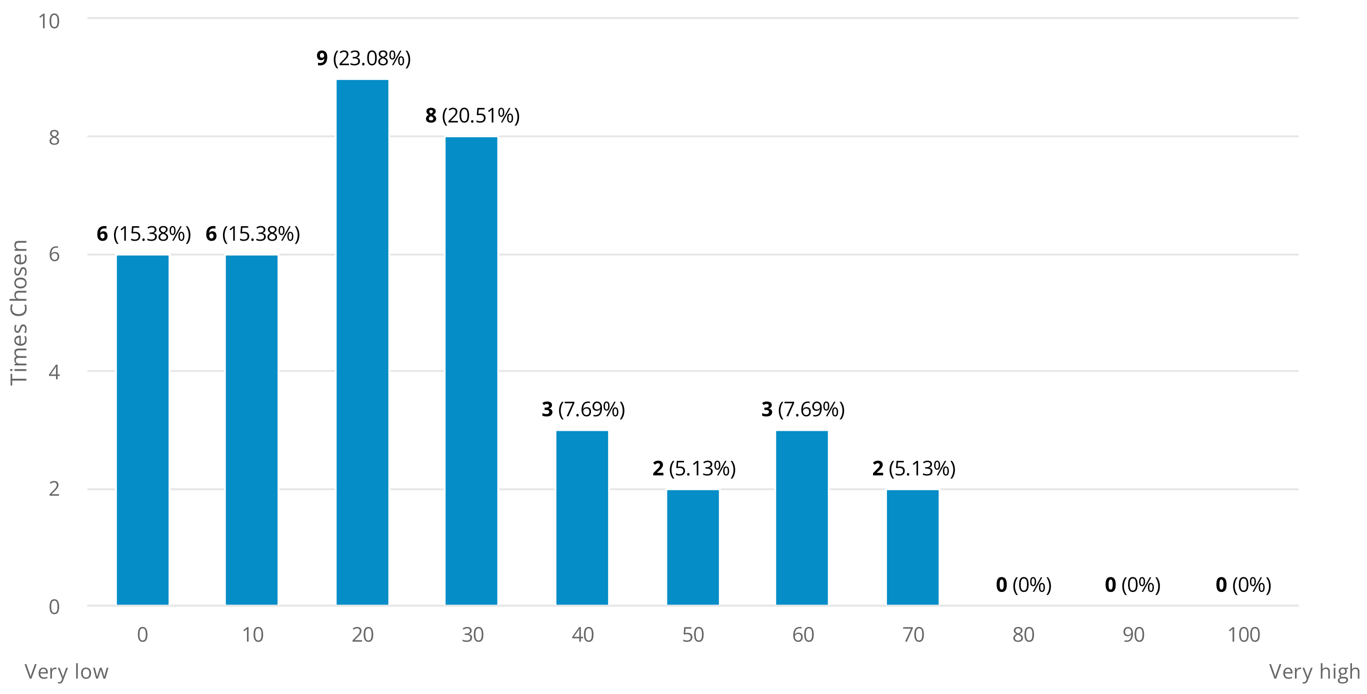
How would you assess the maintainability of the code with design patterns?

Number of responses: 39



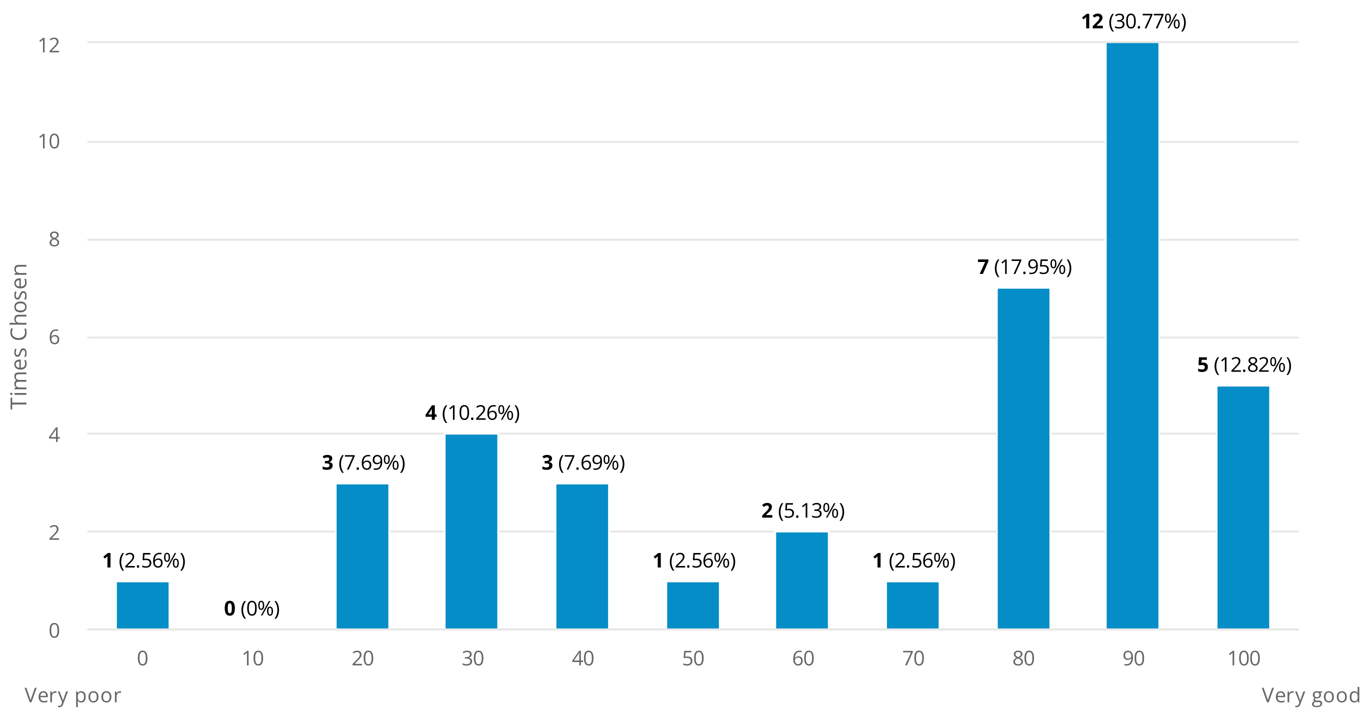
How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)?

Number of responses: 39



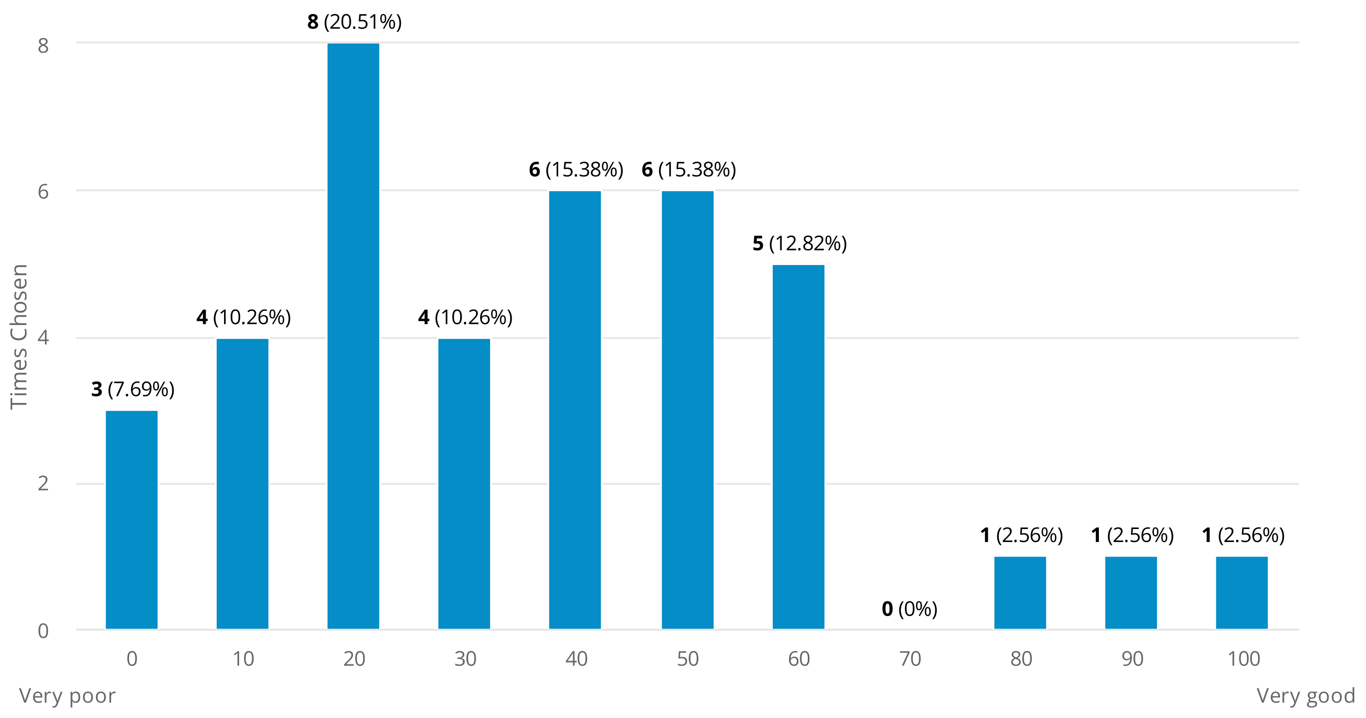
How would you rate the readability of the code in the prototype without design patterns?

Number of responses: 39



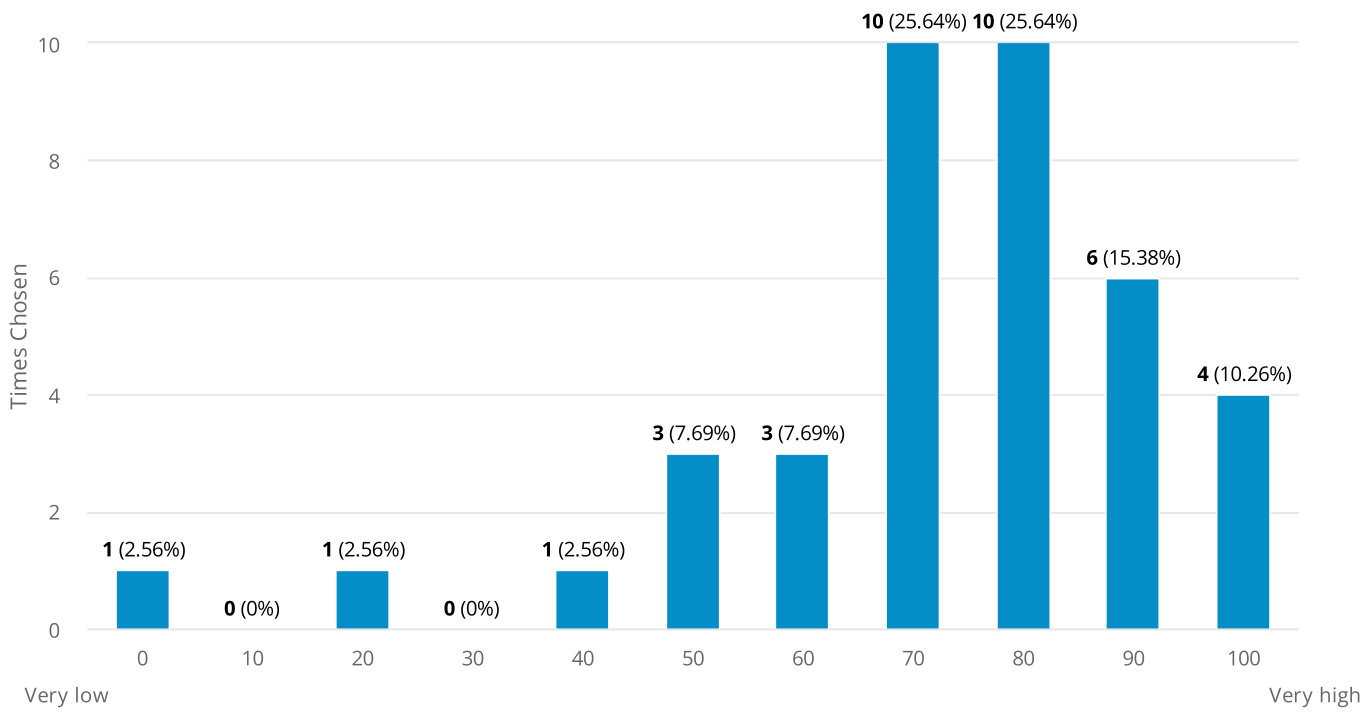
How would you assess the maintainability of the code without design patterns?

Number of responses: 39



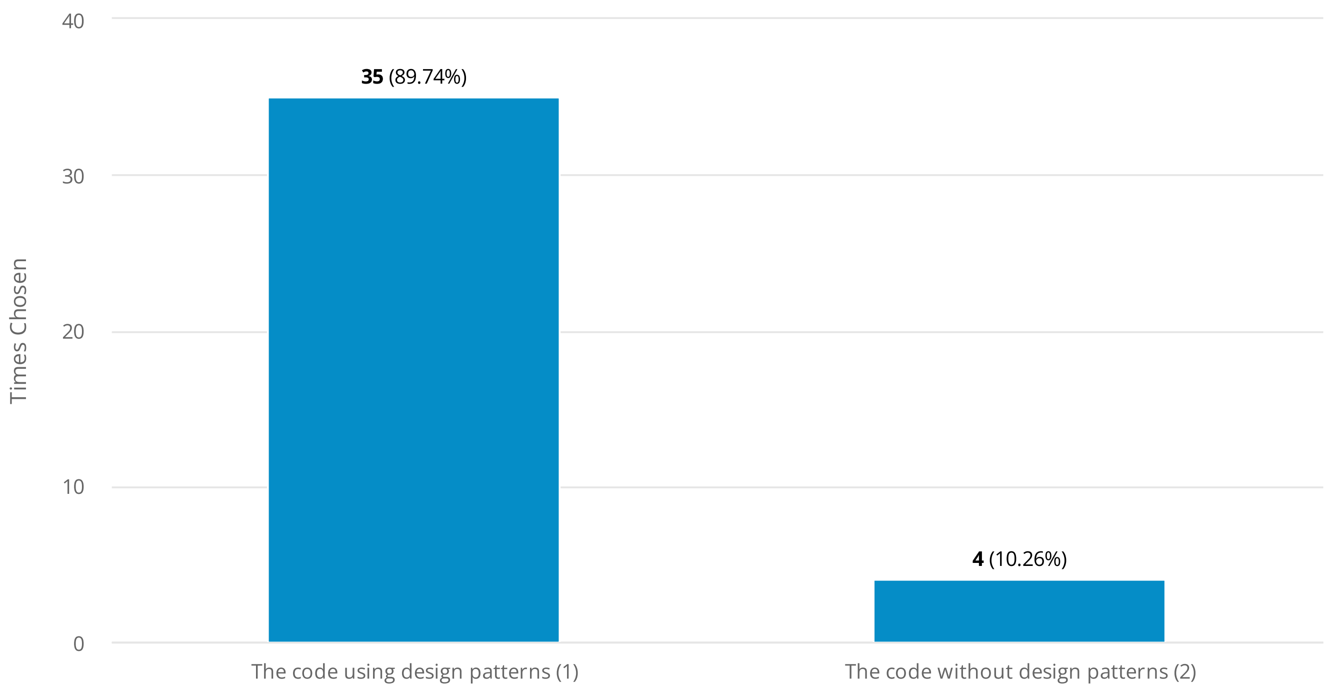
How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)?

Number of responses: 39



Which code would you prefer to work with?

Number of responses: 39



Why would you prefer to work with this code?

Number of responses: 39

Text answers:

because of maintainability and extension

because of better maintainability, single responsebility helps finding the spot where to change or adopt behaviour

keep it simple ist mein Leitsatz beim Design von Code. Speziell in large scale organisations kann es von vorteil sein sich explizit gegen den massigen einsatz von design patterns zu entscheiden. Personen welche den Code zum ersten mal sehen schaffen es in sehr kurzer zeit sich zurecht zu finden.

The readability of the code is much better,

Less code duplication, easier to extend classes and swap them if needed; overall much easier to maintain

Becquse of the better maintainability

Object Oriented code using patterns is convoluted programming stemming from a corporate environment with hundreds of disconnected code monkeys churning out stuff. Of course this is just a personal opinion from someone who would rather maintain decades old Assembler code than touch anything written in Java.

It might be more work to set it up, but I think in the long run it will be much easier to work with and manipulate

familiar pattern w/ trigger name "factory", nice structure & easy to get into w/out pre-knowledge

easyiee

-

It is easier to maintain and in Video Games you oftentimes create similar weapons and change just a bit. So an easy way to so is importanzt

scalability, easy to extend, easy to maintain, re-usability

Weil die Erweiterbarkeit einfacher ist und es flexibler ist

Variant 2 is easier to understand in such a simple example. If the code grows however, it will be much more easy to extend it with more functionality (though I can only guess, what a ShootStrategy looks like).

Using the mentioned design patterns makes it easier to adapt the code to possible changes of requirements

Very good code maintainability, allowing to use it again, no duplicate

particularly due to code duplication issues in code 2

easy to expand, maintain+test

Making change, adding and removing of behaviour easier and clear separation of concern is always good, but implementing exactly one strategy for each weapon might be a bit too much. Having a shoot method in the weapon interface would suffice since there won't be two weapons with the same shooting behaviour. In fact, maybe there could be... different textures/skins, stats and names but same behaviour as in borderlands for example... go with the strategy! 😄

It’s just smaller at first sight and more logical for me

Because at work im crying each day because of replicated code all over the place.

I’d prefer working with the first code because it uses solid design patterns like factory and strategy, making it easier to extend and maintain. It keeps the logic modular and avoids tight coupling, which simplifies adding new weapons or modifying behavior without breaking existing code

Design patterns are simply way more elegant and readble

Although the code using design patterns appears to be harder to read at first glance, it will be way easier to maintain and extend in the future. There will be a much lower chance for duplicate business logic code.

I spent a lot of time in learning patterns. So, I want to use them.

In the beginning, it might be faster and easier to setup code and logic using simple programming that does not utilize any design patterns or strategies.
However, in time, with more features and therefore functionality and logic, the maintenance of those classes will become almost impossible and a refactoring will be required.
So, might as well do it correctly from the start.

Bessere Struktur, bessere wartbarkeit vom Code, bessere Erweiterbarkeit

better for future features

clearer structure, better expandable, and generally easier to work with in a larger scale project such as a Game/Enterprise software in general

Simpler if the codebase is so small as in this example. If it would be a larger project with many Classes, then the design patterns version would probably be the preferred option

Because of the clear structure

Although the code is more with the design pattern and at this simple example harder to read than the code without the pattern, it starts to make a difference in reading, understanding and maintaining the code when the project grows.

I really line splitting code into its desired purpose

For larger projects, definitely design patterns, as it remains clearer.

Cause of the testabilty and maintainbilty of the code. Code with Design Patterns is also easier to extend.

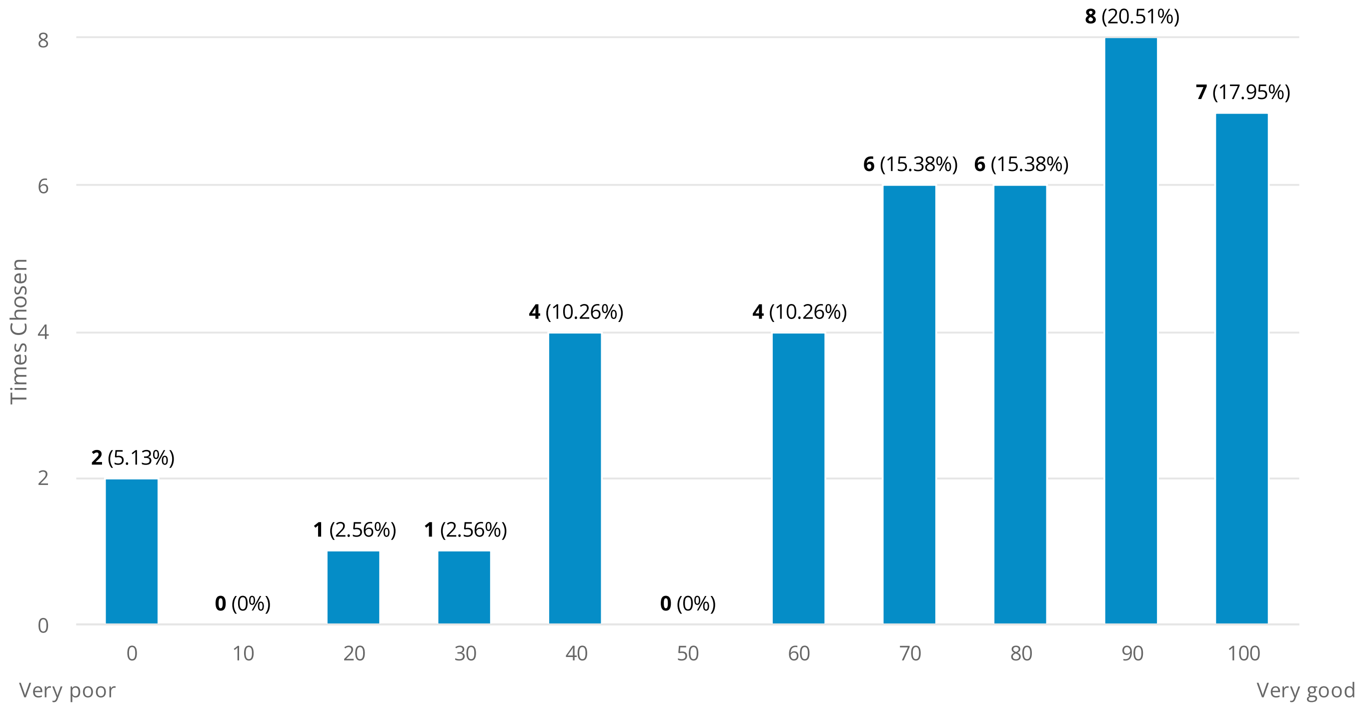
Better structured, easier to look for a error and if using patterns we know its behaviour and when and where to expect errors

It is easier to maintain and more weapons can be added with only little effort

While the code is currently relatively simple to read, even without using patterns, someone unfamiliar with the code would have an easier time understanding the workings of the code in the context of the design patterns; and once all logics are implemented it would be very difficult to follow the train of logic of another developer without the patterns.

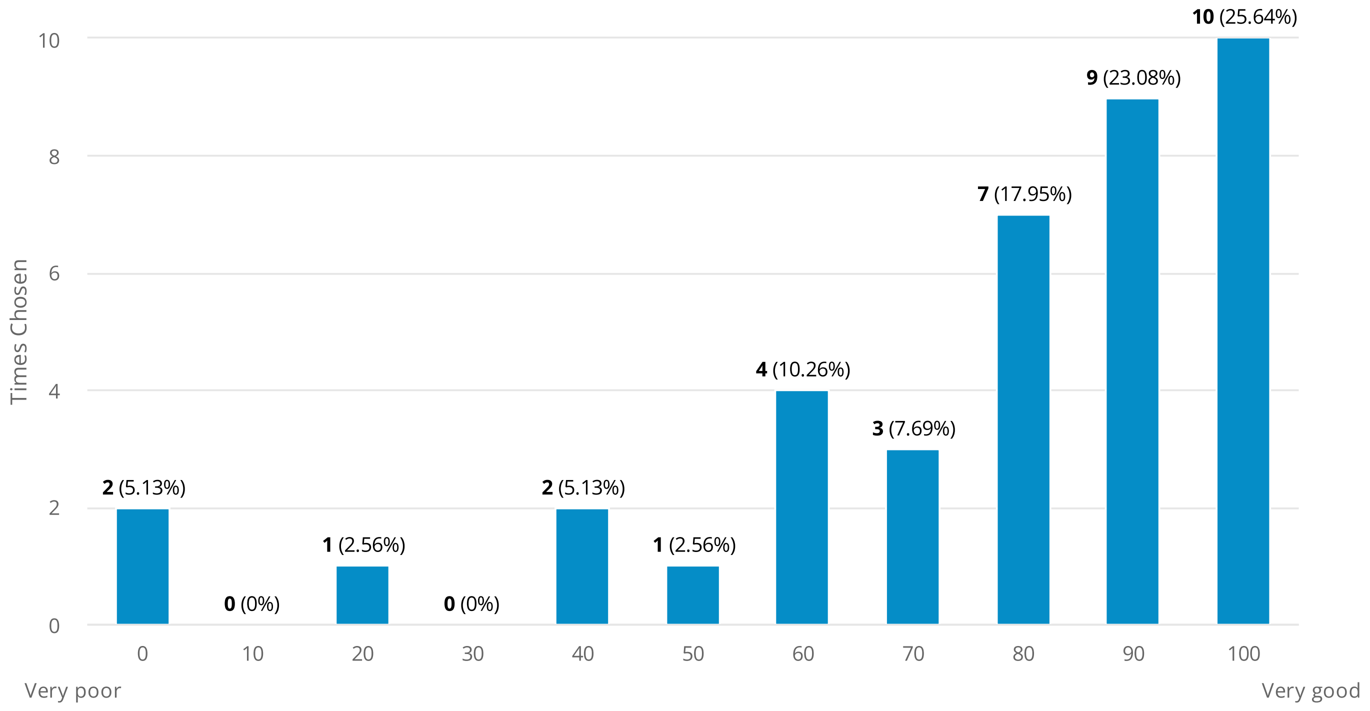
How would you rate the readability of the code in the prototype with design patterns?

Number of responses: 39



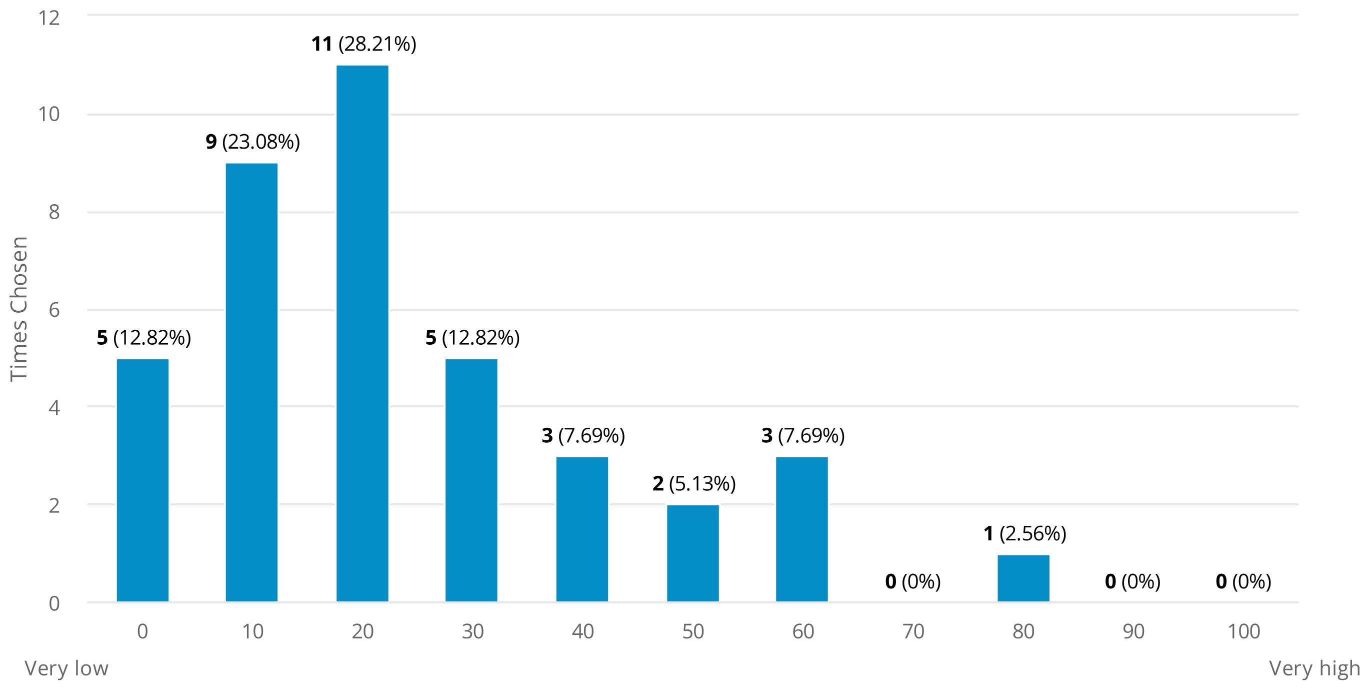
How would you assess the maintainability of the code with design patterns?

Number of responses: 39



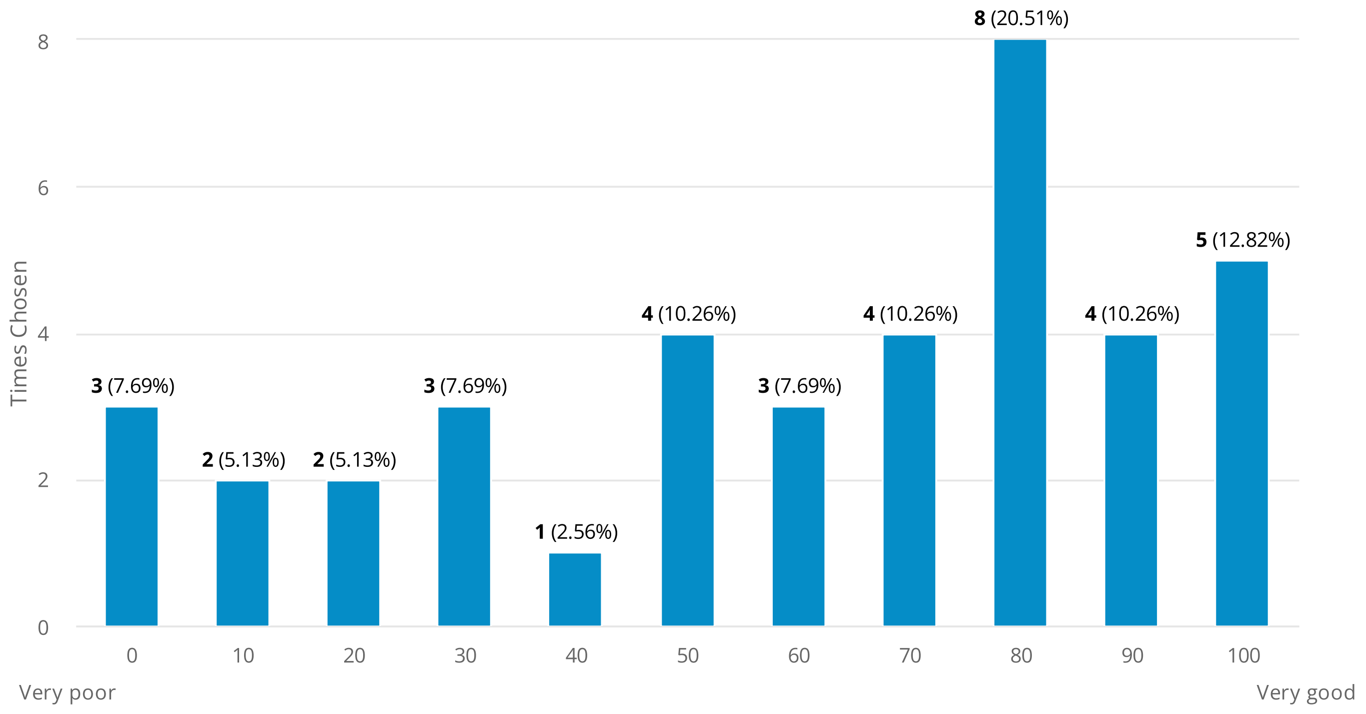
How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)?

Number of responses: 39



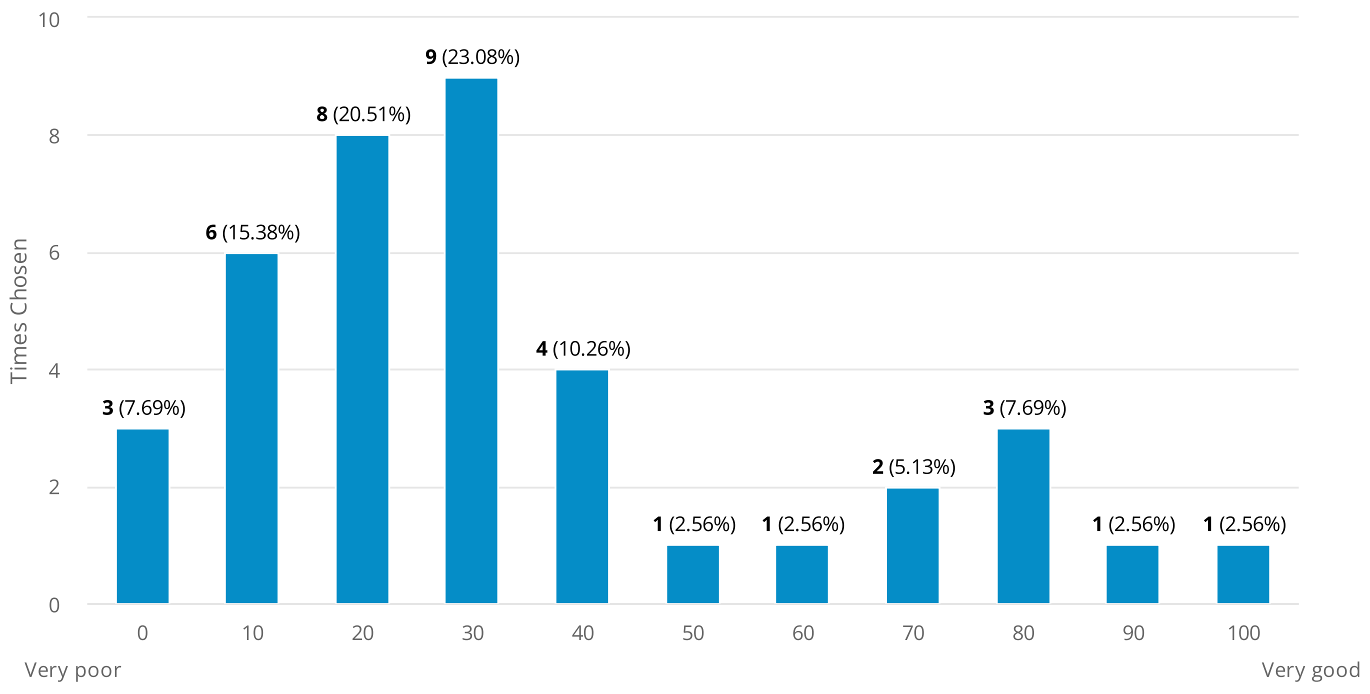
How would you rate the readability of the code in the prototype without design patterns?

Number of responses: 39



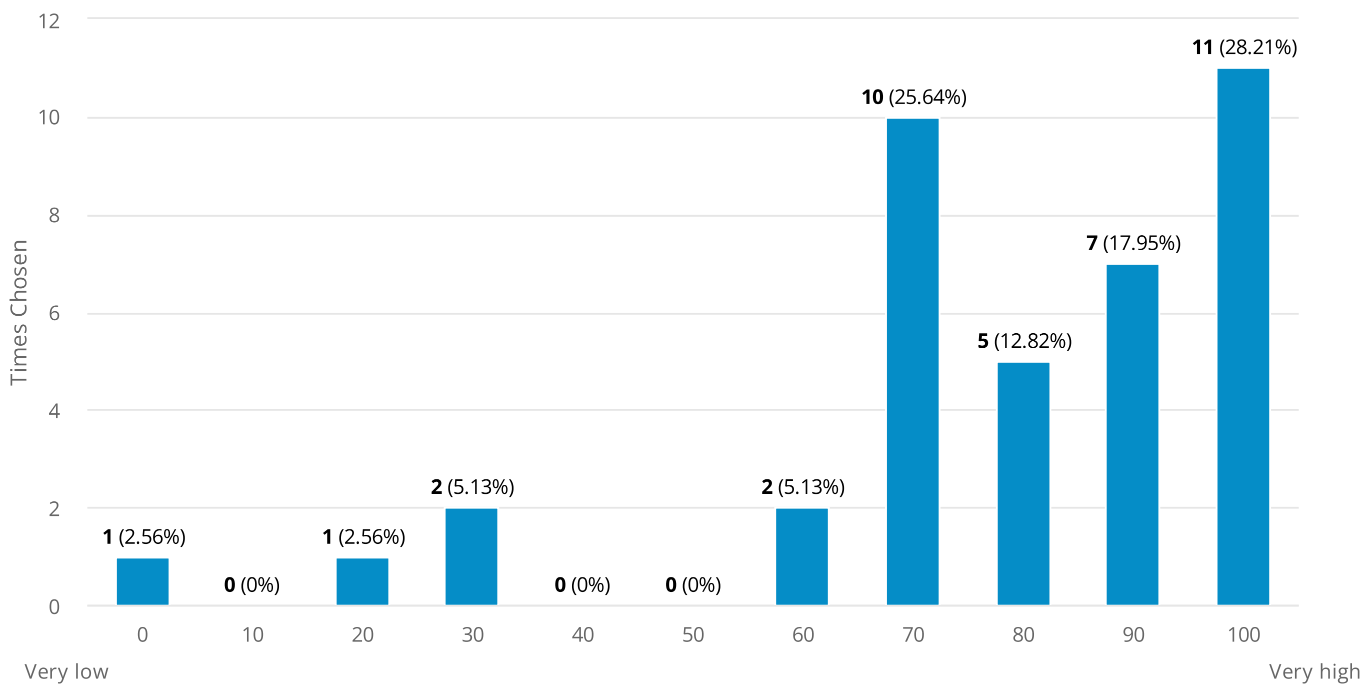
How would you assess the maintainability of the code without design patterns?

Number of responses: 39



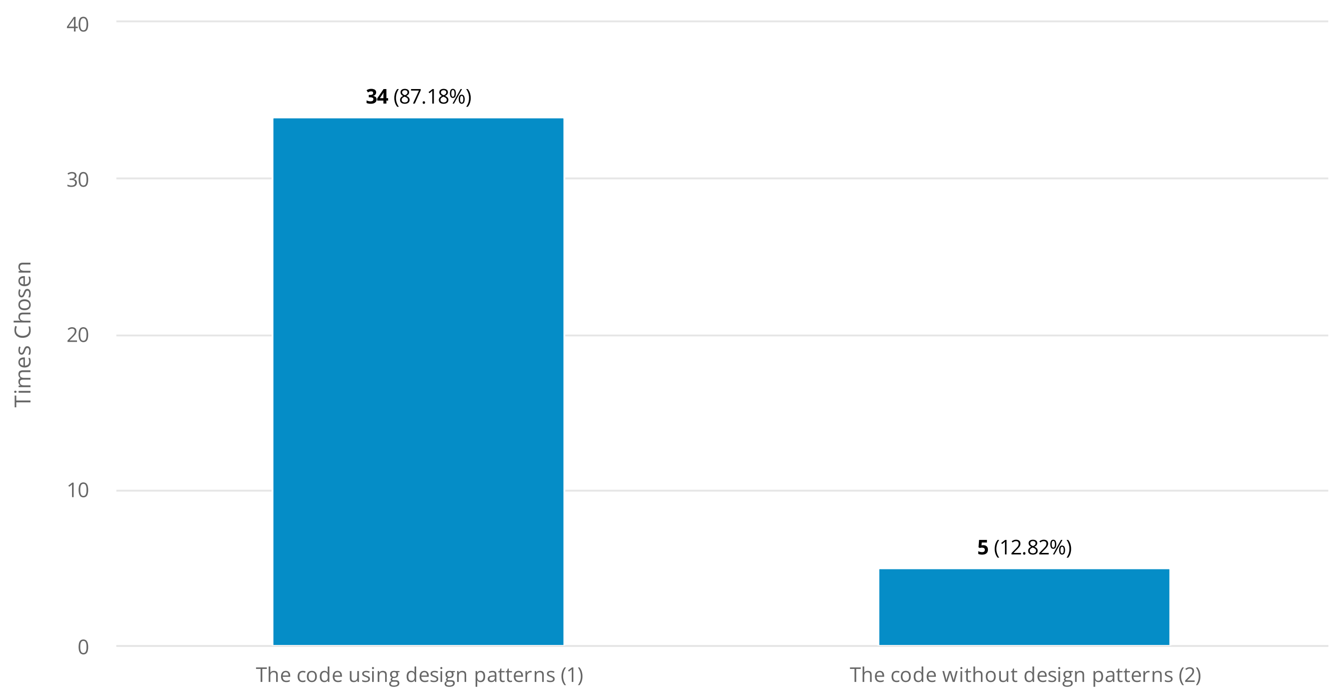
How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)?

Number of responses: 39



Which code would you prefer to work with?

Number of responses: 39



Why would you prefer to work with this code?

Number of responses: 39

Text answers:

because of readability and the risk of code smells

higher maintainability

Siehe antwort zuvor

The readability of the code is much better

Easier to extend the player behaviour, though I'm also not a big fan of the player and state being dependent on each other in the first example (but that's not a big problem considering all the possible annoyances without the pattern)

because of the maintainability

Again .. KISS. Do one thing and do it well. Don't hide functionality in objects where you have to read all the class code to understand what is happening there.

Having one giant player object can lead to a lot of errors if one parameter is changed and is also difficult to maintain

different state in kinda like a separate dictionary (familiarity Python): again easy to understand & read, easy maintain in future (adding/changing states)

/

-

Its easier to extend further and maybe add other controls in other scenarios

easy add more states, handle state changes/update implementation, better testability

Wenn man vorhat ein großes Projekt zu erstellen, sollte man allgemein DPs verwenden. Bei kleinen und einfachen Projekten würde Code 2 nehmen

Again, in such a simple example variant 2 could be sufficient. As one can expect there to be more states when the game grows, variant 1 definitely outperforms variant 2 regarding readability and maintainablity by far. Variant 1 will also most definitely become a god class over time.

Keeping all of the logic in the PlayerController could lead to a bloated class. Adding another type of player state is easier if the state logic is handled in separate classes that implement the same interface.

Make it a very good choice for long term project, and make the code modular and separates concerns

Because code 1 outsources some logic that could also be used by other classes, which reduces code duplication and facilitates maintaining the code.

a bit "clearer"

States get complex pretty quickly and checking for dozens of booleans is cumbersome and prone to errors. better keep them clean and separates as well as possible.

Because for me it’s more intuitive this way

Easier maintenance in the future if code base is million of lines.

Easier to maintain, better overview if more features get added later on

State pattern is the ideal choice for something like player movement

Once again i would prefer the code which uses design patterns because of the more future-proof and extendable design.

A state ist an important part of the code, so it makes sense to use the pattern for more security in handlung the state - just in case.

Similar to the last example.
It really depends also if this class will be further modified in the future.

Bessere Wartbarkeit, Austauschbarkeit etc

better for changes

easier to comprehend and extend

Way cleaner, would be different for a larger project

Clear structure, good readability and easy to extend

extending the functionality is easier and has less risk of breaking the current functionality

Implementing everything in one class makes readability and reuseabulity very narrow making changing the logic impossible

Better maintainability, States can be reused in many classes, so less duplicate Code in different classes

Code using Strategy Pattern is easier to test and easier to extend. I would use the strategy pattern cause it reduces the risk of Large Class and Duplicate Code.

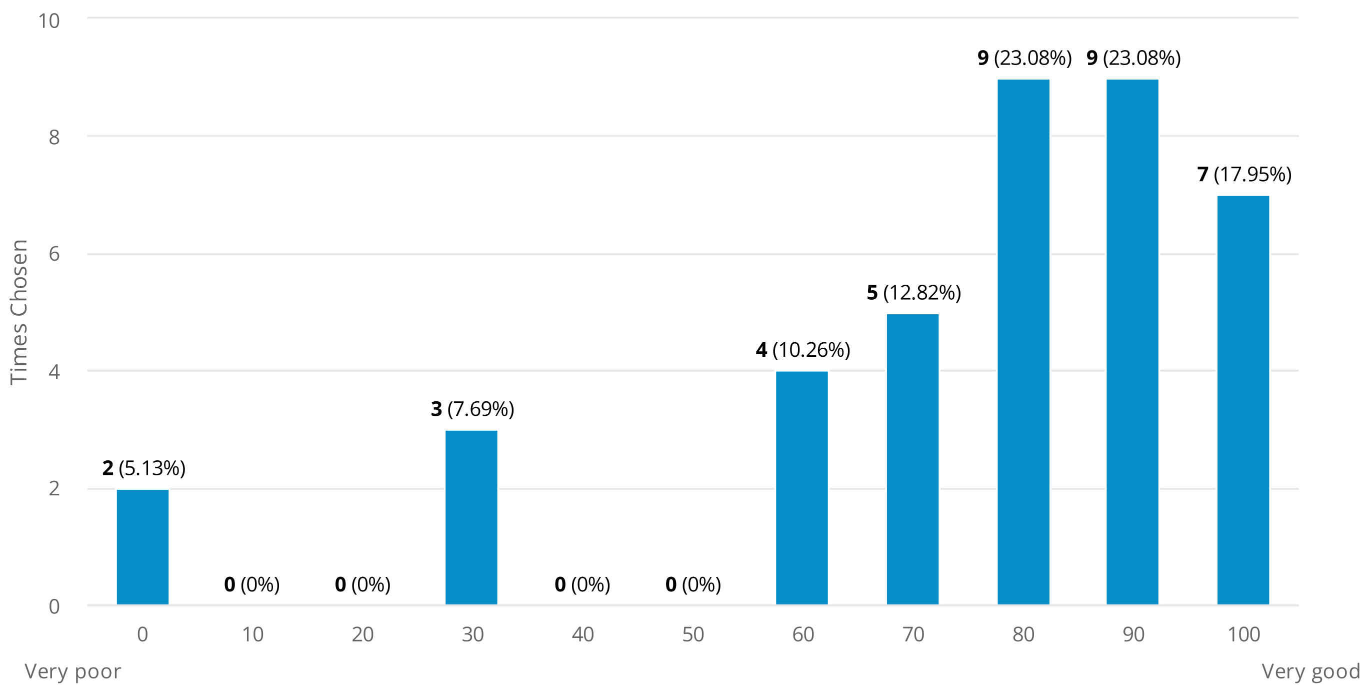
Better readability, less errors

Adding different behaviours is easier

It seems like the work of maintaining the pattern in this case is not worth the benefits gained from using the state pattern in this case.

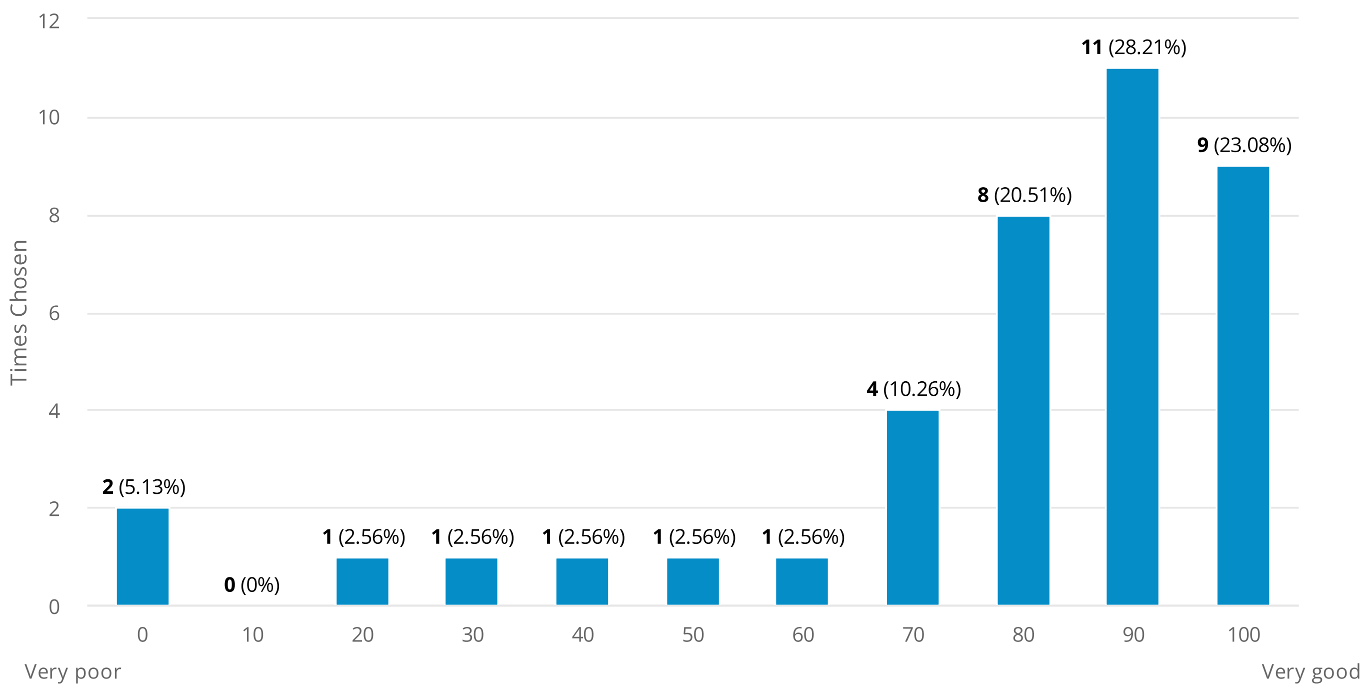
How would you rate the readability of the code in the prototype with design patterns?

Number of responses: 39



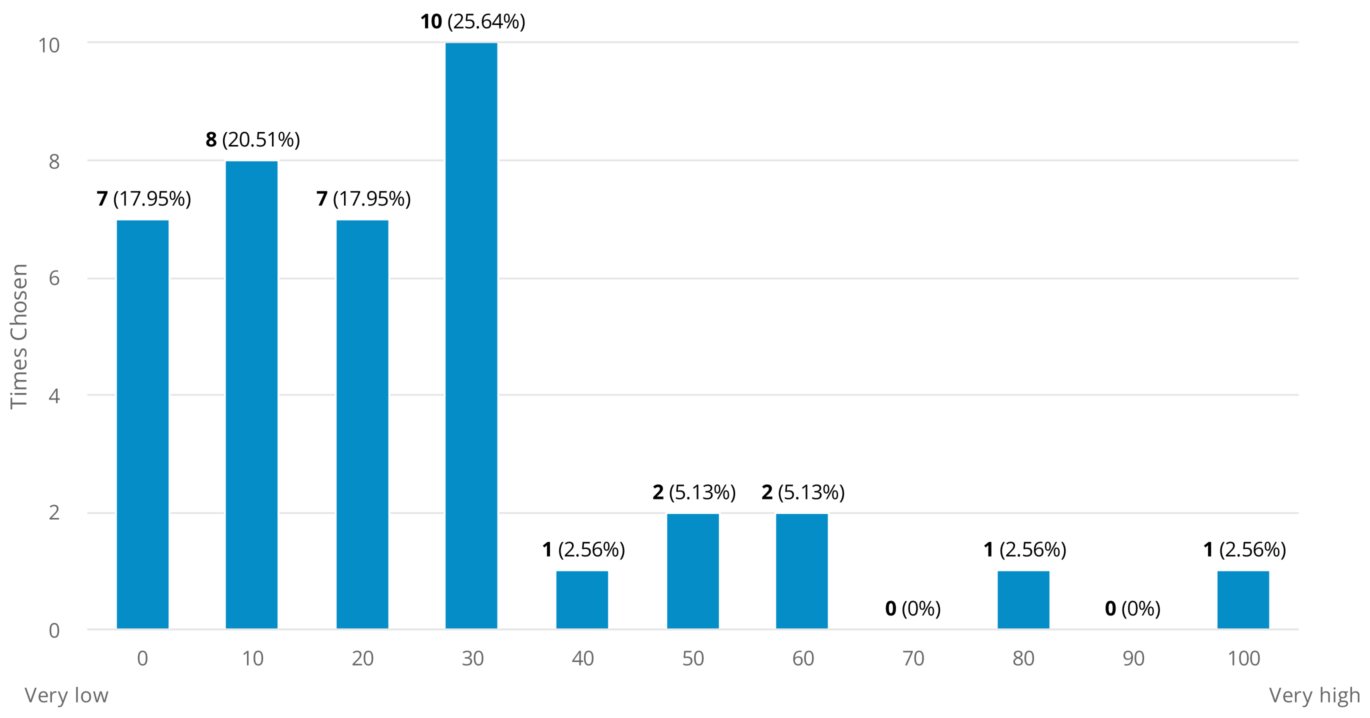
How would you assess the maintainability of the code with design patterns?

Number of responses: 39



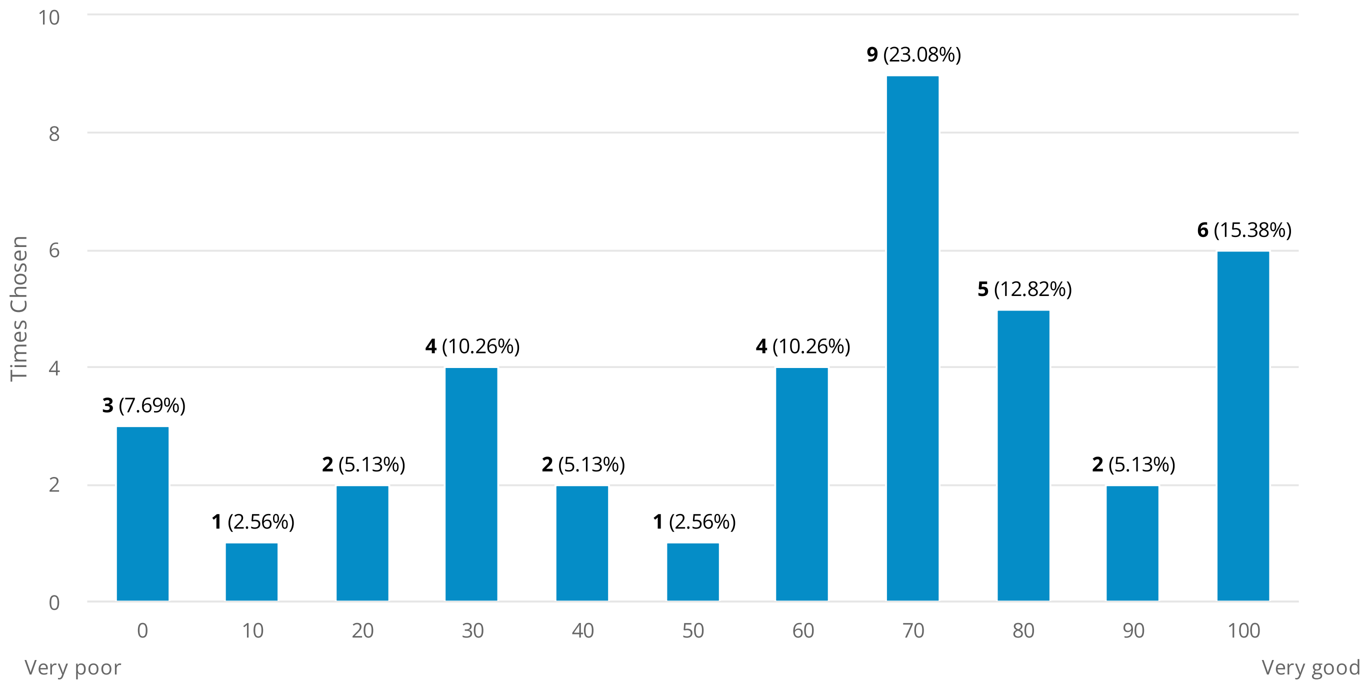
How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)?

Number of responses: 39



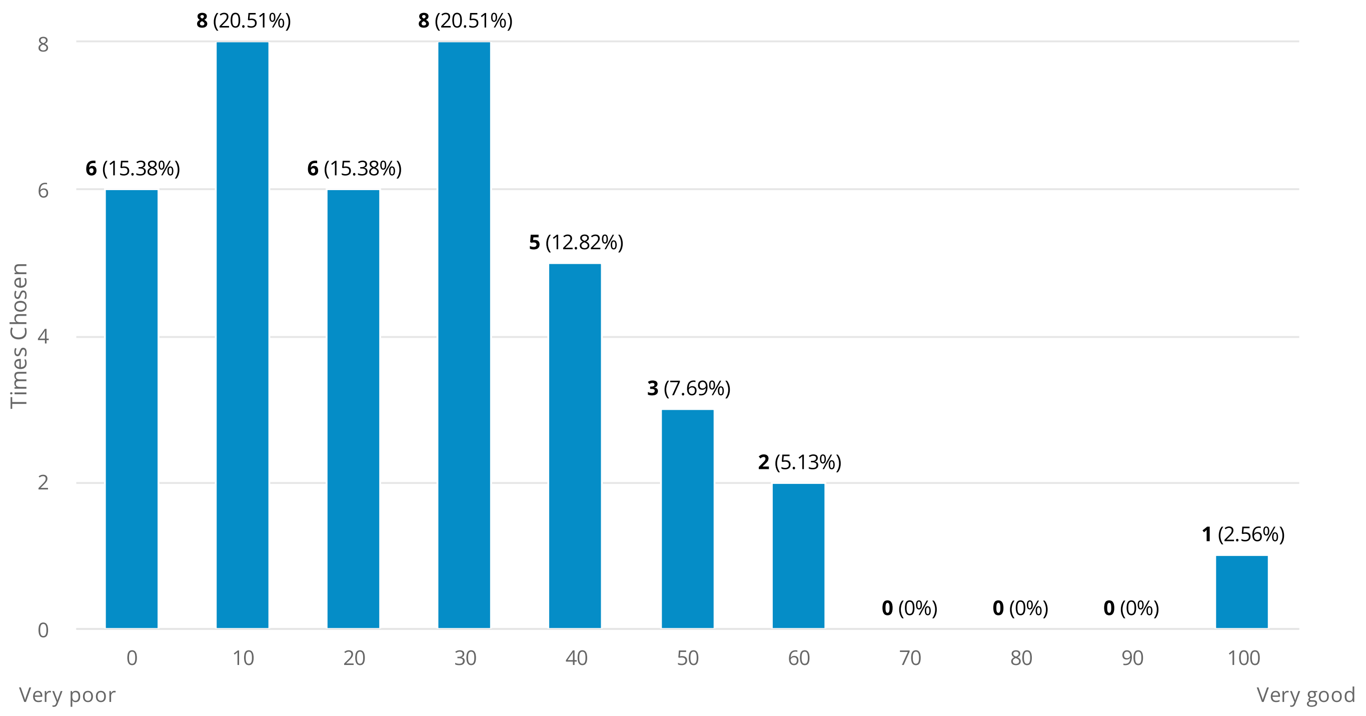
How would you rate the readability of the code in the prototype without design patterns?

Number of responses: 39



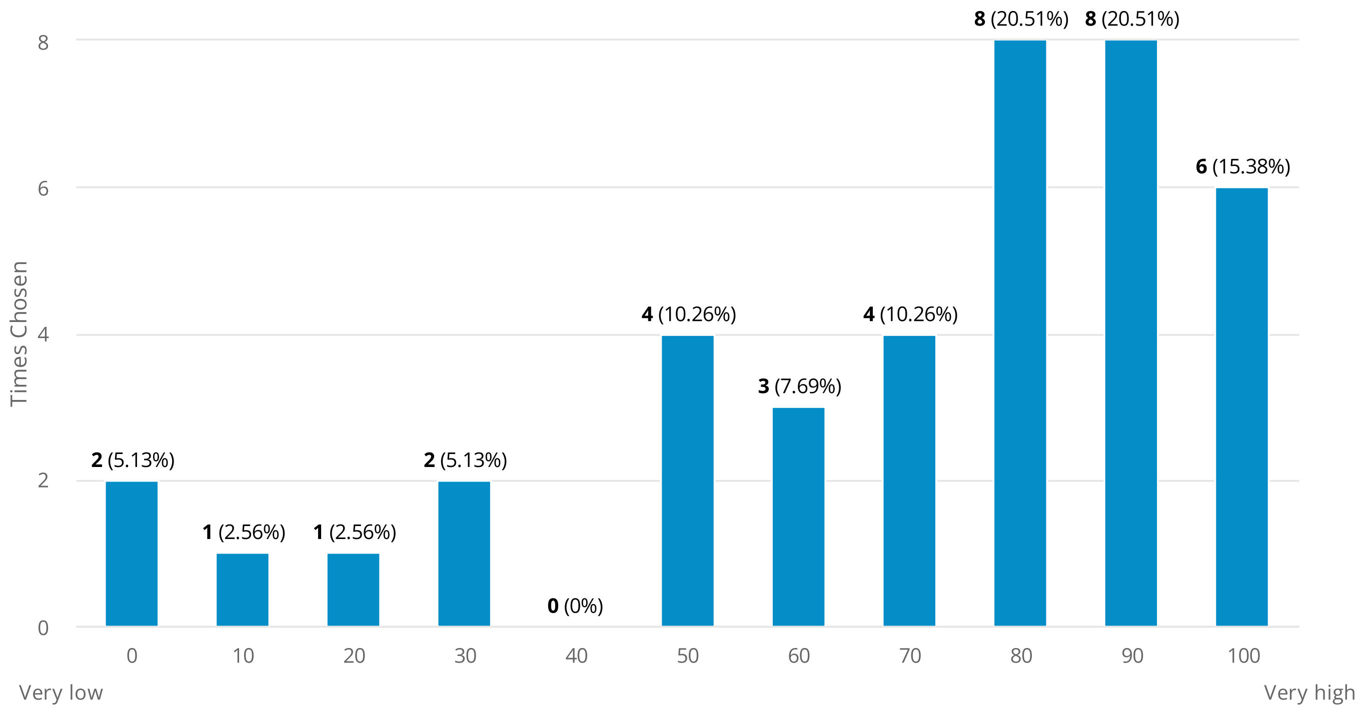
How would you assess the maintainability of the code without design patterns?

Number of responses: 39



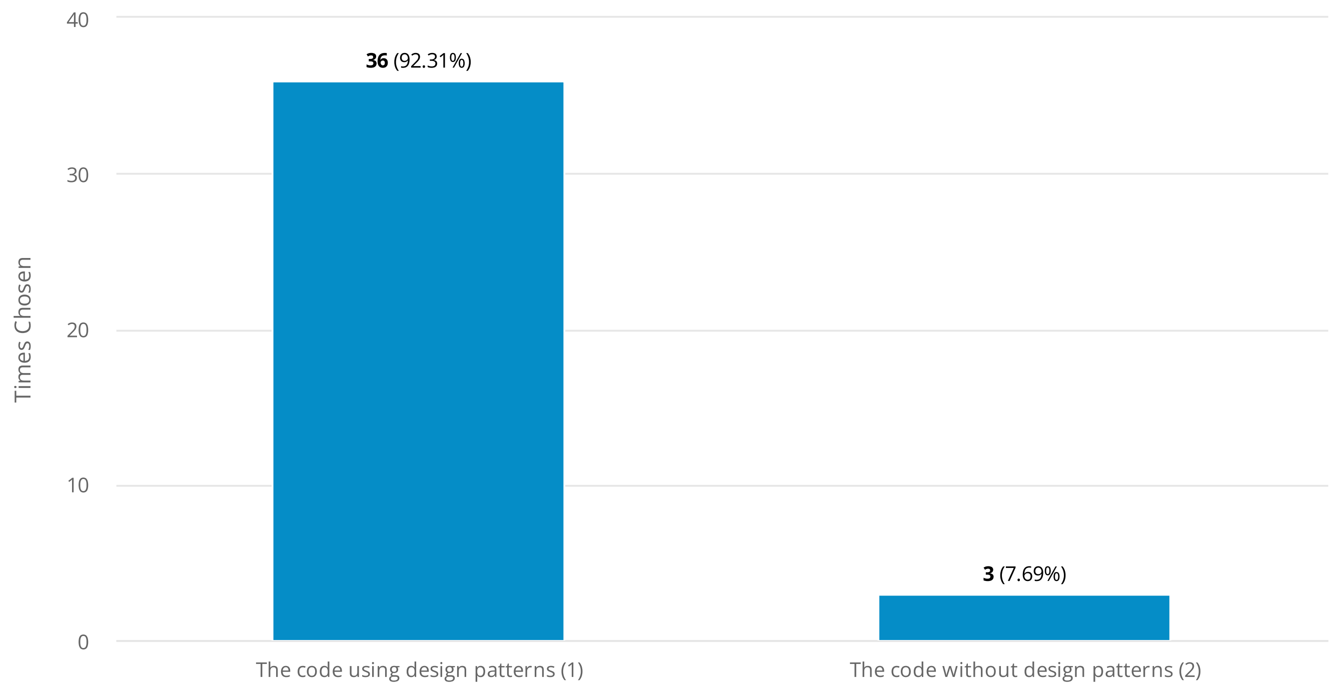
How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)?

Number of responses: 39



Which code would you prefer to work with?

Number of responses: 39



Why would you prefer to work with this code?

Number of responses: 39

Text answers:

more flexible in case of changes

maintainability

Siehe antwort zuvor

The readability of the code is much better

Simpler and very easy to add new functionality, the non-design-pattern one on the other hand, could very easily become a very large class very quickly

maintainability

it's just .. uneccessary complicated and hiding functionality. It's called functionality for a reason, no one wants or needs code to have objectality. Is that even a word?

I think it would get very confusing to always change stuff in the GameManager for every UI change similar reasons (easy understandable & maintaining future)

:

-

Not having the design pattern could also be enough if there are not many of these UIs, but being modular is so much better

testability, easy to extend classes

Weil man das DP einfach verwenden kann und keine unnötigen Code smells entstehen

Same as before, as soon as the code grows, variant 2 will be hard to maintain and extend. In my experience, many objects in a game will listen to the game state, wich will lead to issues in variant 2 over time.

The implementation without design patterns leaves the GameManager with multiple responsibilities. Future changes, like adding additional UI elements, will result in a bloated class. When using the implementation with design patterns, changing UI-specific logic is less likely to affect the GameManager directly.

Making the code easier to follow using observer make the systems more scalable and adaptable for add new features

less ripple effects -> maintainability
Also, the observer pattern nicely encapsulates synchronization logic

adds flexibility, and improves maintainability

Not wanting to use design patterns is like neglecting the architecural achievements of the past millenia and trying to build a building saying "I don't do arches, I don't care about load distribution and I don't like planning - let's just start by piling up stones and see what we get."

In this case I think Pattern 1 is better structured, Pattern 2 can more easy be misinterpreted imo

Easier maintenance for larger code base.

scalable, easier to maintain in larger projects

Alltough the observer pattern ist harder to read and to understand it still is the better choice

And once again I would prefer to use the code using design patterns. I especially like the possibility to add observers at any point in time without needing to change any of the business logic behind.

there is proof that observer are useful - thinking of JS eventlistener.

If a solution uses a design pattern, that makes it also easier to comprehend it as you already know the vague setup of the code.
Especially if the person that implemented it sticked very close to the pattern (even if it is almost impossible to stick to it 100%)

Prevention of codesmells, wartbarkeit

easier to use

easier to maintain
also, I like Observers :)

Allows simpler adjustments across a project

In this case I think the Observer is not necessary. And Observers dont really improve readability and maintainability

readability is very poor without pattern in this case

Each class has their desired functions

Easier to implement new Observers

Observer Pattern reduces coupling of components, therefore the components are easier to test and it is easier to extend the functionality.

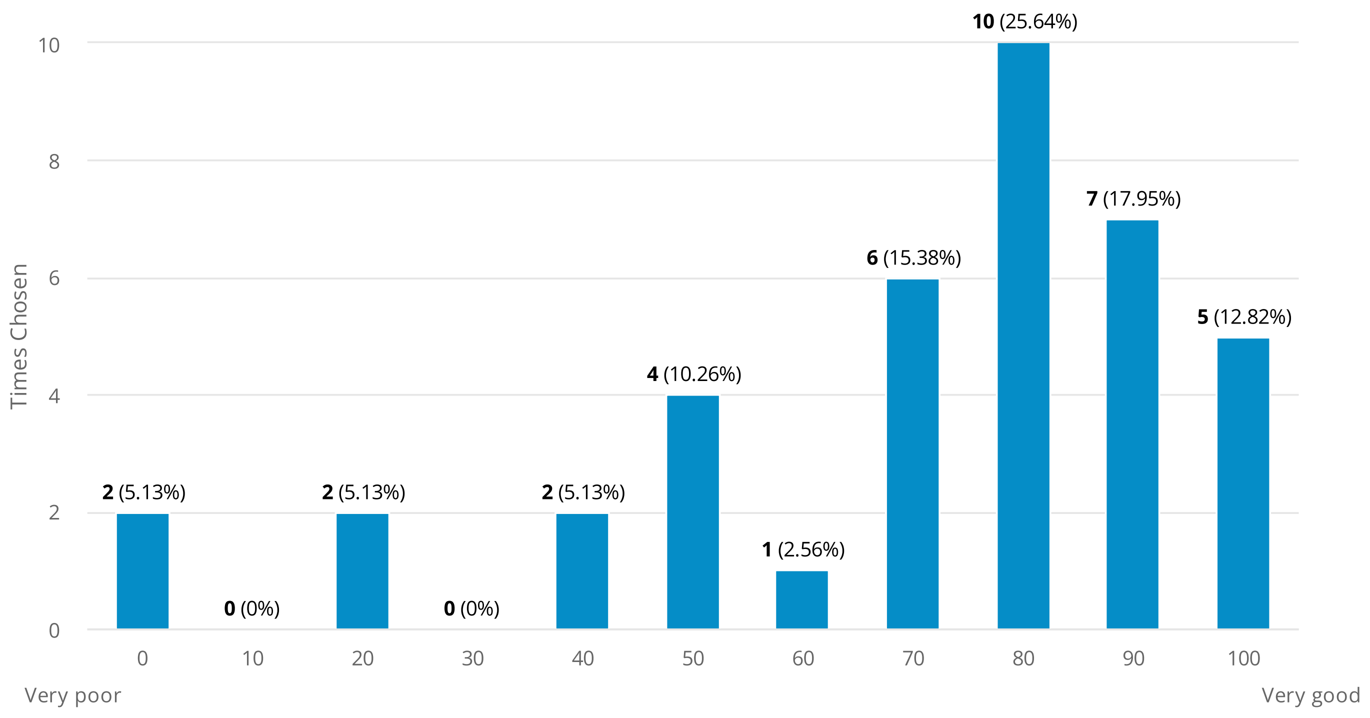
Better readability

At first look it may seem complicated but it is very straightforward and easy to implement

Without the design pattern, the GameManager class would quickly become both a large class and potentially a god class, and the code very hard to maintain not to mention scale.

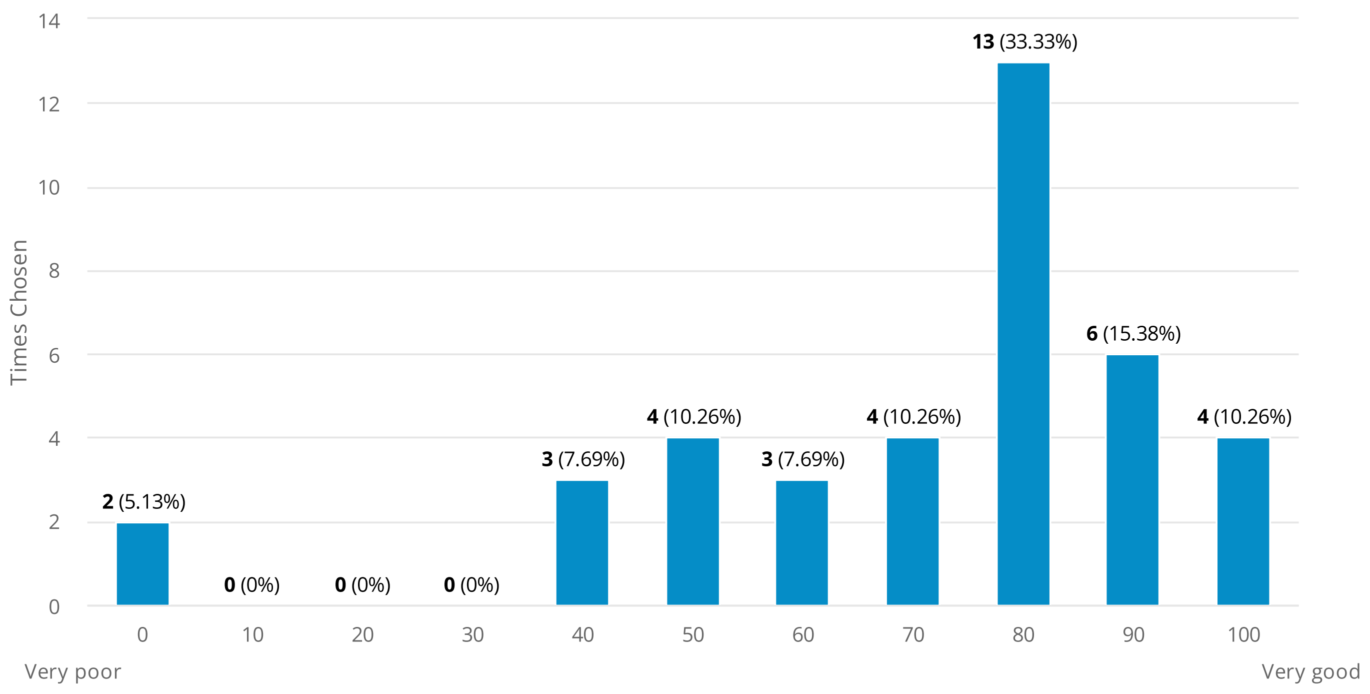
How would you rate the readability of the code in the prototype with design patterns?

Number of responses: 39



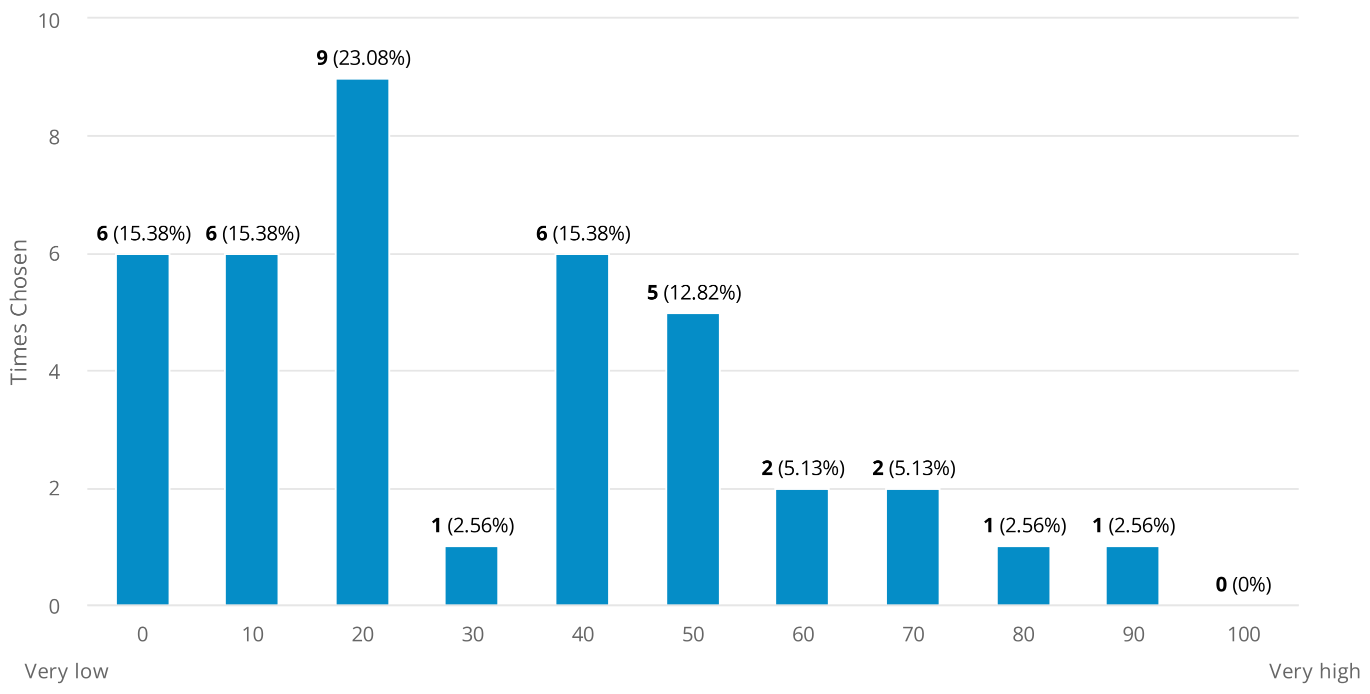
How would you assess the maintainability of the code with design patterns?

Number of responses: 39



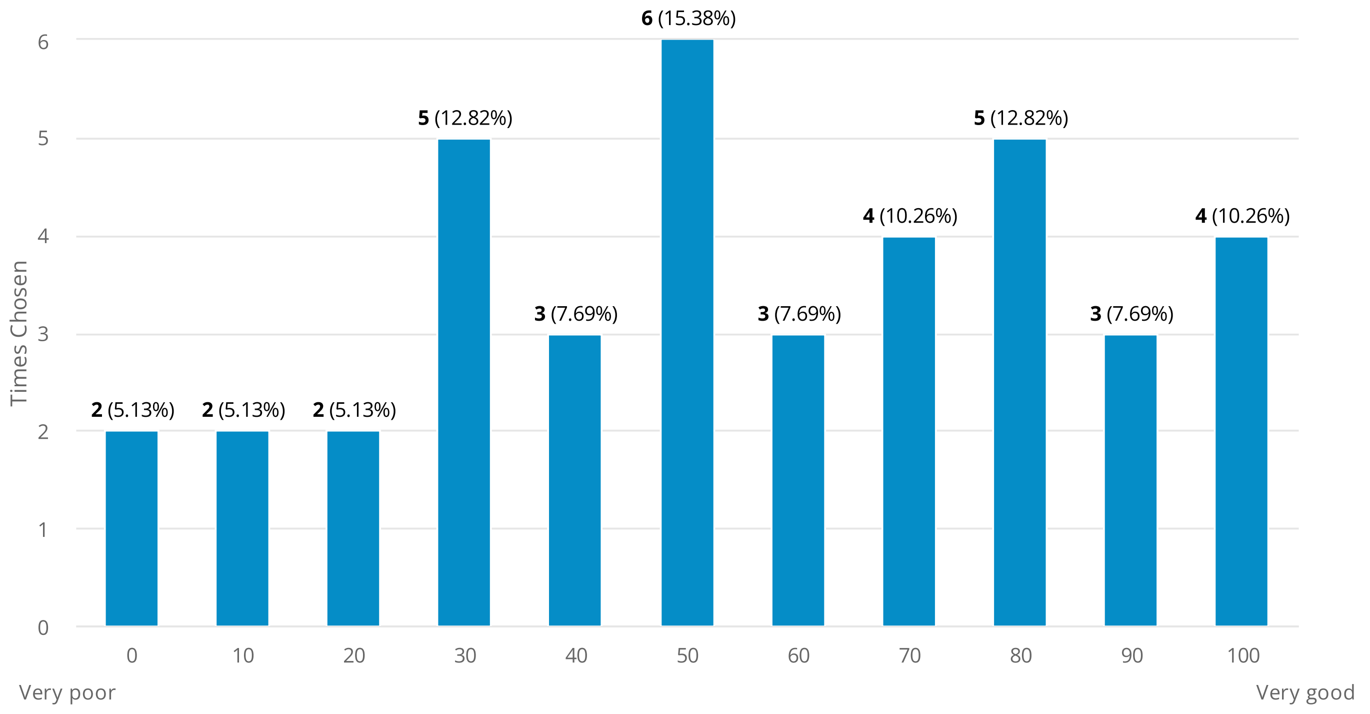
How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)?

Number of responses: 39



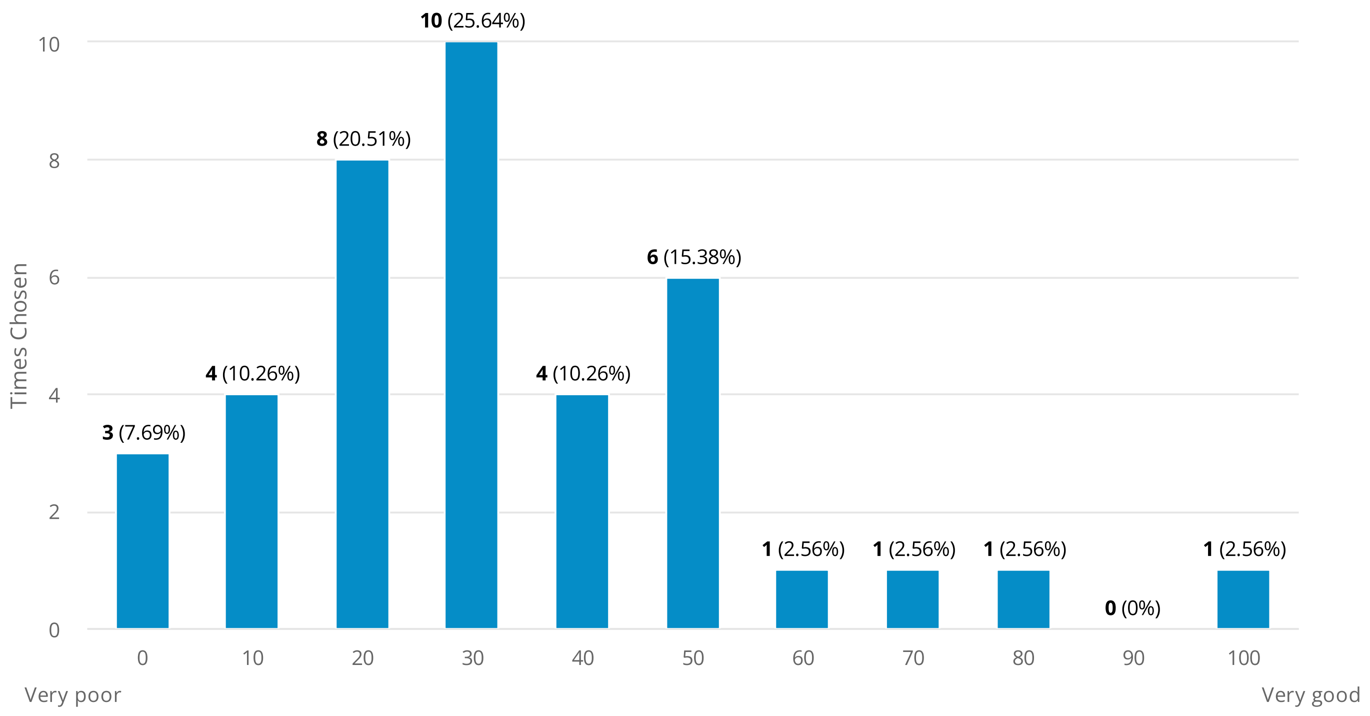
How would you rate the readability of the code in the prototype without design patterns?

Number of responses: 39



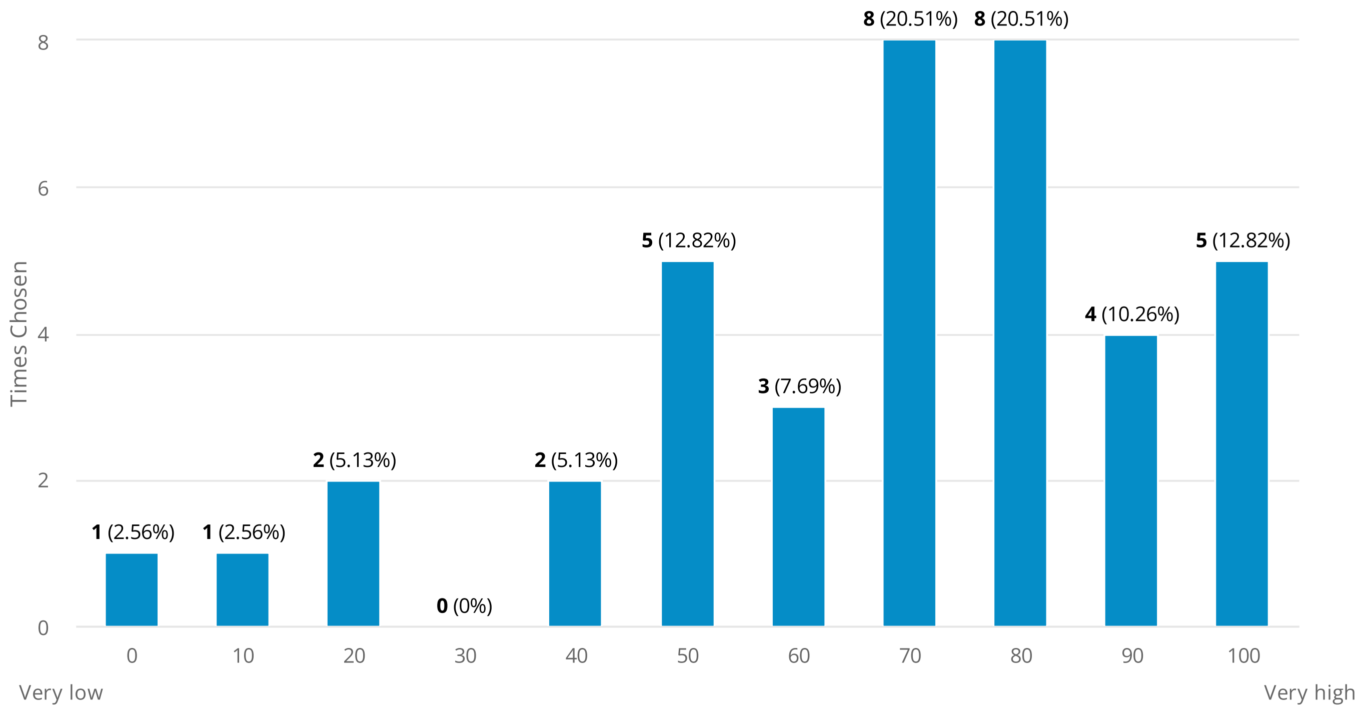
How would you assess the maintainability of the code without design patterns?

Number of responses: 39



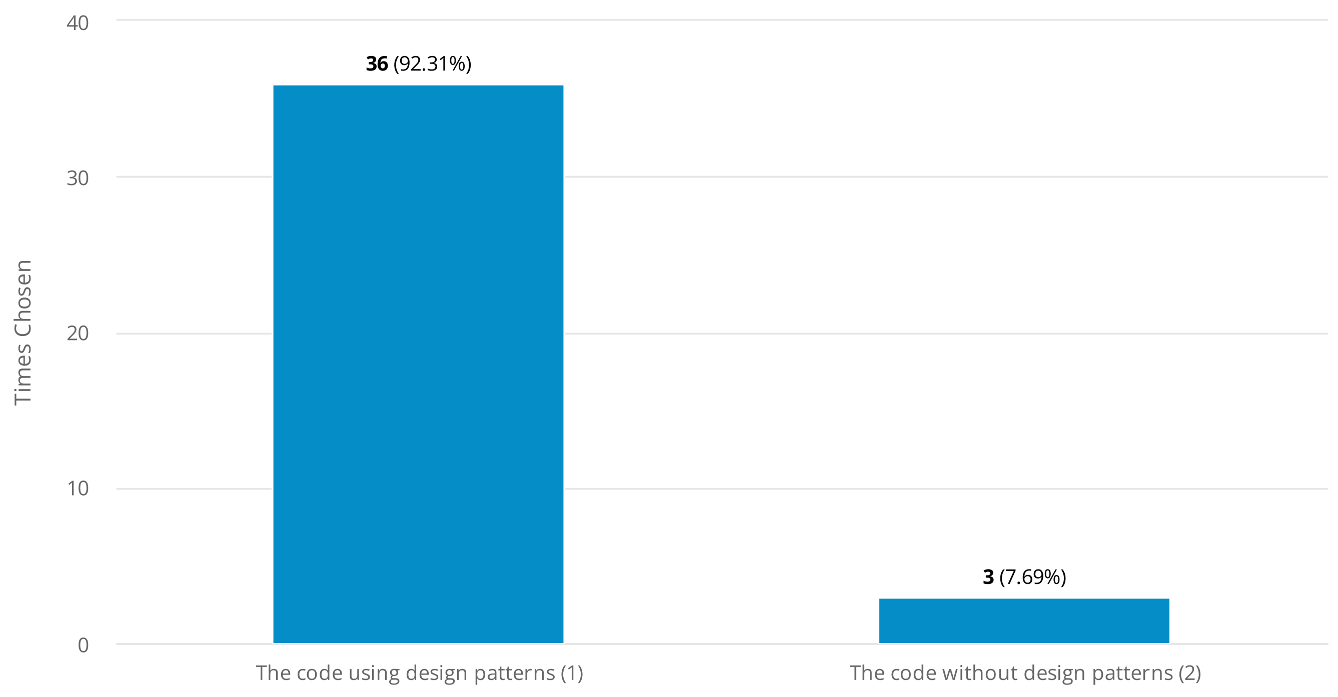
How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)?

Number of responses: 39



Which code would you prefer to work with?

Number of responses: 39



Why would you prefer to work with this code?

Number of responses: 39

Text answers:

because it keeps the access simpler and in general more clear structure

because of centralised logic and shared game states over the project, otherwise you would need to pass states through and thats not good and also a nightmare maintaining it

-

The readability of the code is much better

The singleton pattern is only a small addition and prevents future bugs efficiently, and in this case also has the added benefit of better performance

maintainability

This one is a tie. Singletons are kinda okay, but I'd still be more comfortable without it.

From my experience trying to create a GameManager the Singleton version is very good. Because I did not use one and quickly got overwhelmed with the effort of manipulating things, like creating a new level and adding it to the menu

similar

/

-

Like you said, the alternative to the singleton pattern is less efficient and bad practice.

easy for global access, alternative would be Dependency Injection

Weil das Pattern dafür sorgt das eine Instanz verwendet, dadurch kann es nicht mehrere States, Timer oder Scores haben. Das reduziert die Fehler die das Game bekommen könnte durch duplizierte Scores etc.

Variant 2 might fail if for some reason a second GameManager is instantiated, leading to unintended behavior. Could lead to some strange bugs.

Using the Singleton Pattern provides a single method of obtaining the GameManager instance, instead of having to retrieve it in different ways depending on the context (e.g. manual assignment or using GameObject.Find()).

Easy to read and make the developer don't need to manage references manually or worry about multiple instances

If it is clear that only one instance shall exist, Singleton correctly implements this idea.

simplifies code, might improve performance, better maintainability

I don't recognize the singleton pattern. What does instance = this do? When is the first instance instantiated? Where is the constructor set to private to prevent instantiation? Why would there be another instance that could need destruction?

I prefer #1 because it’s better readable and more logical for me at first inspection

Easier to add more unique instances.

centralized and consistent access to game logic, singleton is a nicer alternative to manually managing references and other more inefficient methods

Singleton is one of the most used patterns due to its security. best choice for single accsess to an instance of something

I would actually prefer to use a singleton in this situation. Since there needs to be one GameManager at any point in time while the game is running, there is no need to take the route of searching and finding an object which is not guaranteed to be unique. For me the singleton is the better choice here. It is probably a bit less maintainable but offers no access overhead and better security in return.

it is always good to know that things should not happen does not happen.

Same as the previous ones.
Especially important to have a design pattern in place for a vital class such as GameState.

Garantiert für eine gute Code Qualität und bessere Wartbarkeit

better controll

easier to follow and comprehend if a singleton is applicable and useful.

Seems to be a bit cleaner, but not sure if it can be a performance bottleneck or if the GameManager would get too large

I think Singleton doesn‘t improv maintainability and scalability but ensures a desired behavior

While in a Singleton Concurrency is hard to manage its still the better option to bundle access in one class if it is needed to be Single source of truth

Better structure better readability

Simplified access, no need to "find" a specific String to get some Data

To Unit-test BallonGoon without Singleton you have to mock GameObject.find, most of the time it is easier to mock your own GameManager as to mock some 3rd-party framework classes like GameObject.

Potential for less errors in code

Simple and effective implementation

Centralising the game logic through a singleton pattern, and having observers use it to update UI elements seems a very elegant solution, which is easy to maintain.

Do you have any additional feedback, suggestions, or thoughts regarding the design patterns or code examples presented in this survey?

Number of responses: 13

Text answers:

The code was hard to read in the survey, I had to open the images in extra tabs and zoom in quite a lot.

Other than that: well written examples in both versions, kudos.

-

No suggestions, but I really like the topic and think its interesting!

SOLID principles are a good way to create clean code and avoid code smells.

No

These questions seem too superficial. Design patterns have been proven helpful over and over again (by definition) but do not avoid code smells by themselves. These questions seem very hard to answer without more context and I think no one in their right mind would choose not using design patterns when building something as complex as a computer game.

none what so ever at this point, looks good

Pretty cool comparison! Good luck for any further work 😊

The benefits of many patterns only get visible when the code base is/gets larger

It was a bit hard for me as I wasn‘t familiar with the programming language. I think it was C#?

prototypes are fine without patterns but when a projects scales its important to be clear about how to extend functionality savely

I'm not sure if the last example with the Singletn pattern is compareable. Looks like as they were two different examples.

No ! Very well done