# Software Development Principles

# What are they?

- They serve as best practices when developing software
  - General guidelines and there can be exceptions

- Without them you might still have working code, but may have difficulties with:
  - Debugging
  - Testing
  - Reading
  - Maintenance

# Software Maintenance

- Can be used to reference development activities that are not purely maintenance (like new features)

- A more modern term is Software Evolution
  - A wholistic view of the process of software development
  - Considers not just activities to keeping existing software running, but also expanding features

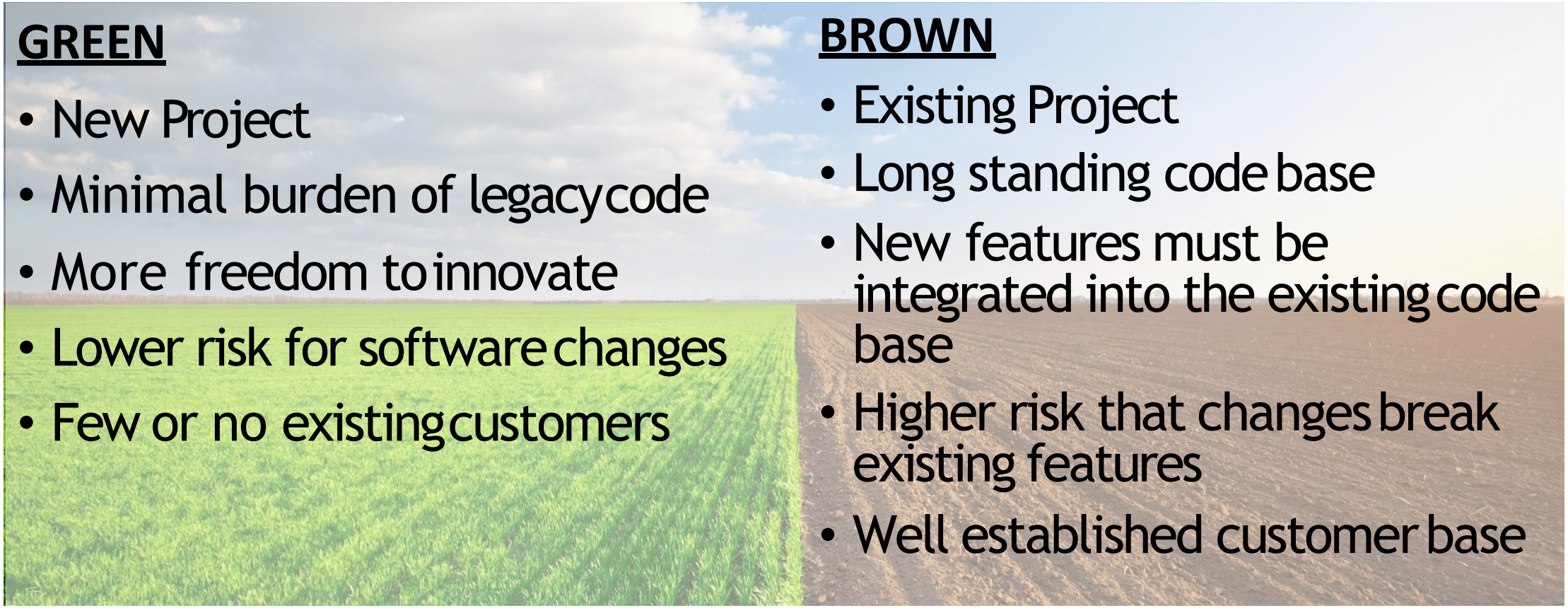
- Maintenance accounts for 80% of software development costs [US Department of Commerce 2002]

# Why is maintenance so expensive!?

- Software products vary drastically in size
  - https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

- If your software is used by people…it will inevitably require changes
  - OS updates
  - New versions of programming languages or features
  - Library changes
  - Bugs
  - New features driven by the market or users

# Greenfield vs. Brownfield



# Separation of Concerns (SoC)

- Code that is concerned with one thing should go in one place
  - Network communication, reading and writing files, accessing a database, all the code that deals with statistics, etc.
- In C this happens at the function level and the module level
  - Modules should contain related functions address the same overall concern
- Easier to test units of code
  - Functionality is isolated
- Easier to maintain
  - more likely that a change will only have to happen in one place

# Don't Repeat Yourself (DRY)

- If you have the same code in multiple places in your program, it may be best to put it in a function
- Makes it more likely that a change will only have to happen in one place
- Having duplicate code in your program is sometimes referred to as a "code clone"
- These are classified as "codesmells"
  - Things that may be a potential design issue

# KISS (Keep it simple, student)

- Simpler is often better
- A simple function is easier to understand than a complex function
- "Clever" solutions can be hard to understand
- Simple code is easier to maintain

# Refactoring

- Changing the structure of your code without changing how it behaves
- There are lots of refactoring patterns
  - https://refactoring.com/catalog/
- Often done before adding a feature or fixing a bug
- Example
  - Problem: A program uses a for loop in multiple locations to print the contents of an array
  - Solution: Extract the for loop and printing to a function and call that when necessary

# What about optimizations and performance?

- Everyone likes ”fast” programs…
- However, performance matters most when a program it too slow to accomplish a task or does not meet requirements
- Be careful not to try and optimize too early
- Prioritize readability and maintainability

# Comments and Documentation

- Comments are good when they focus on the purpose of the code
  - People can read what code does but might not know why it's important
  - If it needs a clarification comment, it might be overly complicated

- Code can also be "self-documenting"
  - `int foo(int x, int y)` **VS.** `int average(int sum, int count)`
  - `int x` **VS.** `int total_pay`

- Function documentation helps people understand how to use your code

# Invariants

- Things that are always true (or should always be true)

Function invariants:

- Preconditions
  - What is true about the inputs
  - This is what the function expects from the caller
- Postconditions
  - What is true about the output
  - Return values, things that are printed, communication over the internet

# Function Invariants Example

```
/*
   Preconditions:
   array is an array with size number of elements

   Postconditions:
   The return value is the sum of the elements.
*/
int array_sum(int array[], size_t size);
```

# Useful Tips

- Build up your code slowly
  - Don't try to write the whole program and once and then compile, run, and test
- Compile frequently
  - Typos happen and your compiler is the first line of defense (think of it like an advanced spell check)
- Test your code incrementally
  - The compiler doesn't catch everything, you need to run the code
  - Think about what types of input your program might get and see if the results are what you expect