1. QR Codes

In this example, the `open_from_qr_code` method both creates a QR code scanner object and then uses that scanner to obtain the URL to navigate to. This makes the `Browser` class harder to test because now you'd have to patch the `QRScanner` object in order to test `open_from_qr_code`. Also, it's not possible to replace the QR code scanner with another scanner (for example, one that uses another camera than the main front or back camera of the device).

You can solve this by moving the creation of the `QRScanner` out of the `Browser` class and pass it as an argument instead:

```python
class Browser:
    def open(self, url: str) -> None:
        print(f"Opening {url} in the browser.")

    def open_from_qr_code(self, qr: QRScanner) -> None:
        qr.choose_camera(Camera.BACK)
        url = qr.scan()
        self.open(url)

def main() -> None:
    print("Navigating to website on device.")
    browser = Browser()
    qr = QRScanner()
    browser.open_from_qr_code(qr)
```

What you see is that in the updated version of the code, creating objects now happens in a single place: the `main` function. This is great, because it gives you as a developer a lot more control over when and where resources are created since all of that is managed in a single place. This is what makes the Separate Creation From Use principle so powerful!

2. Game Enemy Factory

a) The Abstract Factory is going to be responsible for spawning enemies. The easiest way to set this up is to create an abstract base class that defines the interface for the factory, as follows:

```python
from abc import ABC, abstractmethod

class EnemyFactory(ABC):
    @abstractmethod
    def spawn(self) -> Enemy:
        pass
```

Now we can create subclasses of `EnemyFactory` that implement the `spawn` method. For example, the `EasyEnemyFactory` class can be implemented as follows:

```python
class EasyEnemyFactory(EnemyFactory):
    def spawn(self) -> Enemy:
        enemy_type = random.choice([EnemyType.KNIGHT, EnemyType.ARCHER])
        health = random.randint(30, 60)
        attack_power = random.randint(20, 40)
        defense = random.randint(10, 20)
        return Enemy(enemy_type, health, attack_power, defense)
```

Similarly you can define the `MediumEnemyFactory` and `HardEnemyFactory` classes. See the full code in the `solution2.py` file.

b) A more functional approach to creating enemies based on the type of spawn point would be to use a dictionary that maps the spawn point type to a function that creates the enemy. For example:

```python
def easy_spawn() -> Enemy:
    enemy_type = random.choice([EnemyType.KNIGHT, EnemyType.ARCHER])
    health = random.randint(30, 60)
    attack_power = random.randint(20, 40)
    defense = random.randint(10, 20)
    return Enemy(enemy_type, health, attack_power, defense)

def medium_spawn() -> Enemy:
    # code for medium spawn

def hard_spawn() -> Enemy:
    # code for hard spawn

class SpawnType(StrEnum):
    EASY = "easy"
    MEDIUM = "medium"
    HARD = "hard"

SPAWN_FUNCTIONS = {
    SpawnType.EASY: easy_spawn,
    SpawnType.MEDIUM: medium_spawn,
    SpawnType.HARD: hard_spawn,
}
```

Now you can use the dictionary to create enemies based on the spawn point type:

```python
def spawn_enemies(spawn_type: SpawnType, count: int) -> list[Enemy]:
    spawn_function = SPAWN_FUNCTIONS[spawn_type]
    return [spawn_function() for _ in range(count)]
```

For the complete version of this more functional approach, see the `solution2_fun.py` file.