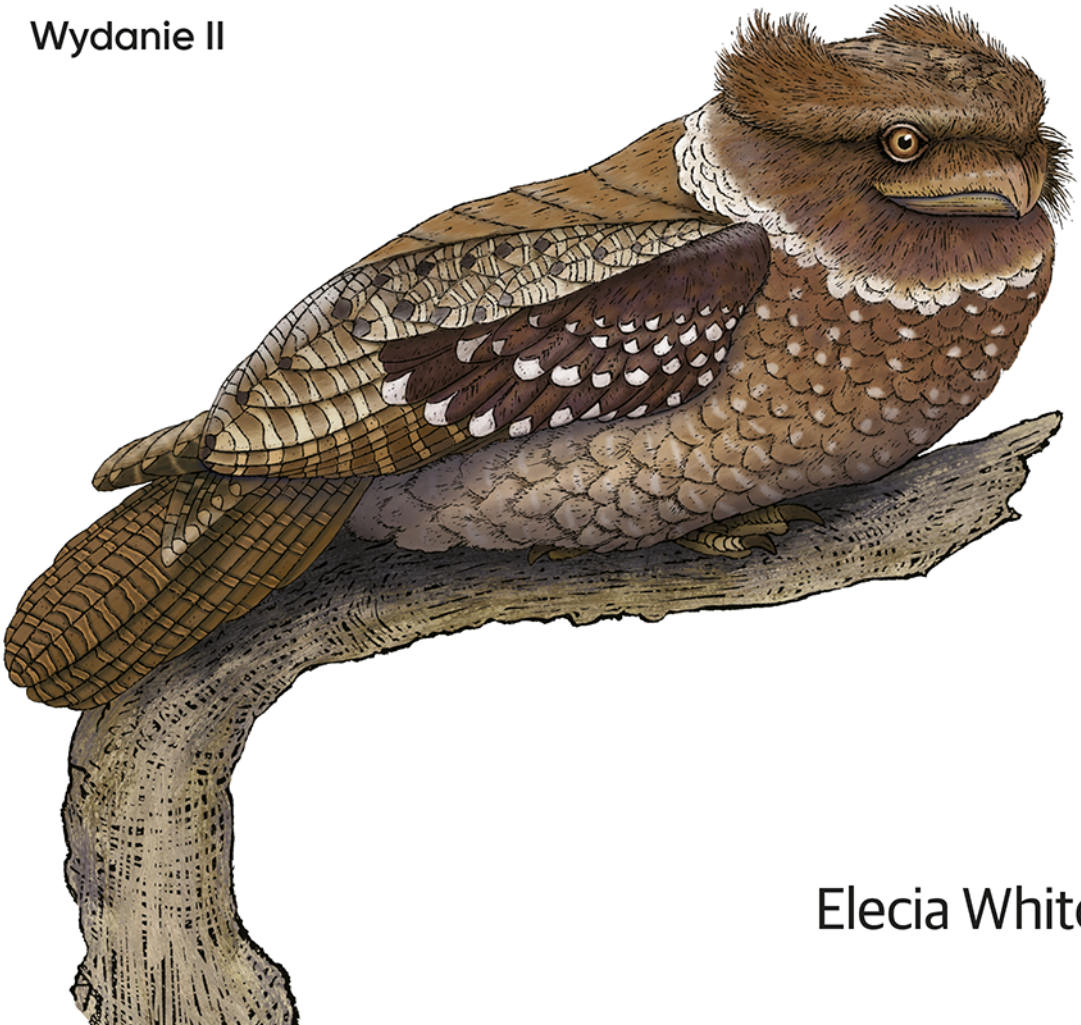
O'REILLY®

Helion

Systemy wbudowane

Wzorce projektowe dla twórców oprogramowania

Wydanie II



Elecia White

Tytuł oryginału: Making Embedded Systems: Design Patterns for Great Software, 2nd Edition

Tłumaczenie: Grzegorz Werner

ISBN: 978-83-289-1829-0

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/syswb2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/syswb2.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Tworzenie architektury systemu

Nawet małe systemy wbudowane mają tyle szczegółów, że trudno rozpoznać w nich miejsca nadające się do zastosowania wzorców. Będziesz potrzebował dobrej panoramy całego systemu, żeby zrozumieć, które części mają proste rozwiązania, a które ukryte zależności. Dobry projekt powstaje przez zaczęcie od takiego sobie projektu i stopniowe ulepszanie go, najlepiej przed przystąpieniem do implementacji. A diagram architektury systemu jest dobrym sposobem na to, aby przyjrzeć się systemowi i zacząć projektowanie oprogramowania (albo zrozumieć plątaninę przekazanego Ci kodu).

Definicja produktu rzadko jest ustalona od samego początku, więc możesz zacząć od rozważania różnych pomysłów. Kiedy funkcje produktu zostaną naszkicowane na tablicy, możesz przystąpić do myślenia nad architekturą oprogramowania. Ludzie od sprzętu będą robić to samo (oby w kontakcie z Tobą jako projektantem oprogramowania, choć mogą się skupiać na nieco innych rzeczach). Nie minie wiele czasu, a będziesz mieć architekturę oprogramowania i wstępny schemat. W zależności od Twojego poziomu doświadczenia kilka pierwszych projektów, nad którymi będziesz pracować, prawdopodobnie będzie opartych na innych projektach, więc sprzęt będzie bazował na istniejącej platformie z pewnymi zmianami.

W tym rozdziale omówimy różne sposoby patrzenia na system pod kątem projektowania lepszego oprogramowania wbudowanego. Stworzymy kilka diagramów opisujących nasz system i sugerujących sposoby projektowania architektury oprogramowania. Diagramy pozwalają nie tylko lepiej zrozumieć system, ale również przygotować oprogramowanie na zmieniające się wymagania poprzez enkapsulację, ukrywanie informacji, tworzenie dobrych interfejsów między modułami itd.

Systemy wbudowane są w dużej mierze zależne od sprzętu. Niestabilny sprzęt sprawia, że oprogramowanie wydaje się wadliwe i zawodne. Właśnie od tego trzeba zacząć: upewnić się, że będzie można oddzielić zmiany w sprzęcie od usterek w oprogramowaniu.

Można też pójść w drugą stronę, czyli przyjrzeć się funkcjom systemu i ustalić, jaki sprzęt jest potrzebny do ich obsługi (co zwykle jest preferowane). Ja jednak skupię się głównie na niskopoziomowym oprogramowaniu do łączenia sprzętu, aby podkreślić ideę, że sukces Twojego produktu zależy od stabilności i dostępności funkcji sprzętowych.

Kiedy projektujesz „od dołu do góry”, jak opisano tutaj, traktuj sprzęt jako archetypowy. Choć ostatecznie będziesz musiał poznać specyfikę sprzętu, początkowo przyjmij, że istnieje

jakiś sprzęt, który spełnia Twoje wymagania (tzn. jakiś procesor, który robi wszystko, czego potrzebujesz). Wykorzystaj go do opracowania architektury systemu, oprogramowania i sprzętu, zanim zagłębisz się w szczegóły.

Pierwsze kroki

Do projektowania systemu można zabrać się na dwa sposoby. Jednym jest zaczęcie od czystej karty, z pomysłem na efekt końcowy, ale bez żadnych gotowych elementów. Jest to **kompozycja**. Tworzenie diagramów systemu od podstaw bywa zniechęcające. Pusta kartka może się wydawać ogromna.

Drugim sposobem jest przejście od istniejącego produktu do architektury, czyli wzięcie czegoś, co złożył w całość ktoś inny, i zbadanie wewnętrznych elementów. Taka inżynieria wsteczna to **dekompozycja**, dzielenie rzeczy na mniejsze części. Kiedy dołączasz do zespołu, czasem zaczynasz od napiętego terminu, odrobiny kodu i bardzo ograniczonej dokumentacji, bez czasu na napisanie dodatkowej.

Diagramy pomogą Ci zrozumieć system. Znajomość jego ogólnej struktury pomoże Ci napisać Twoją część systemu w sposób, który zapewni wysoką jakość kodu i pozwoli dotrzymać terminu.

Możesz rozważyć stworzenie dwóch diagramów architektonicznych: jednego lokalnego dla kodu, nad którym pracujesz, i bardziej globalnego, który opisuje cały produkt, żebyś mógł zobaczyć, jak wpasowuje się w niego Twoja część.

Im więcej zrozumiesz, tym większe będą się stawać Twoje diagramy, szczególnie w przypadku dojrzałego projektu. Jeśli rysunek stanie się zbyt skomplikowany, połącz kilka elementów w jeden i rozwiń je na osobnym rysunku.

Istnieją różne rodzaje rysunków, a niektóre z nich lepiej nadają się do opisania konkretnego systemu; zależy to od tego, jak pierwotny architekt myślał o tym, co robi.

W kilku następnych podrozdziałach dla prostoty poproszę Cię o narysowanie kilku diagramów od podstaw, bez przyglądania się istniejącemu systemowi (wszystkie dobre systemy są na tyle skomplikowane, że nie mogłabym pokazać ich w tej książce).

Pokazuję je jako szkice, ponieważ taki właśnie jest zamysł. Różne diagramy to różne spojrzenia na system oraz to, jak można (powinno się?) go zbudować. Nie tworzymy dokumentacji systemu; szkice te są sposobem na myślenie o systemie z różnych perspektyw.

Twoje szkice powinny być na tyle niedbałe, że nie poddasz się, jeśli będziesz musiał zrobić je od nowa. Jeśli chcesz, użyj programu do rysowania, ale moją ulubioną metodą jest skorzystanie z papieru i ołówka. Pozwala mi to skupić się na informacjach, a nie na metodzie rysowania.

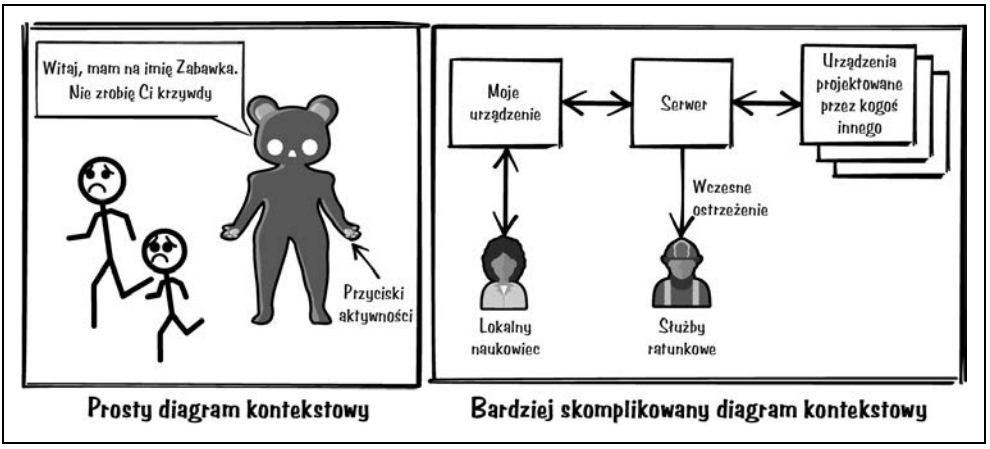
Tworzenie diagramów systemu

Podobnie jak projektanci sprzętu tworzą schematy, powinieneś sporządzić serię diagramów, które pokazują relacje między różnymi częściami oprogramowania. Takie diagramy zapewniają panoramę całego systemu, pomagają identyfikować zależności i zapewniają wgląd w projekt nowych funkcji.

Polecam cztery typy diagramów: diagramy kontekstowe, diagramy blokowe, organigramy i diagramy warstwowe.

Diagram kontekstowy

Pierwszy rysunek pokazuje, jak Twój system wpisuje się w szerszy kontekst. Jeśli jest to dziecięca zabawka, diagram może być prosty: ludzik z kresek i kilka przycisków na zabawce. Jeśli jest to urządzenie gromadzące informacje z sieci czujników sejsmicznych i dodatkowo pokazujące dane lokalnym naukowcom, system jest bardziej skomplikowany i wykracza poza część, nad którą pracujesz (rysunek 2.1).



Rysunek 2.1. Dwa diagramy kontekstowe z różnymi poziomami złożoności

Celem jest tu przedstawienie ogólnego kontekstu, w jakim system będzie używany przez klienta. Nie martw się o to, co znajduje się w poszczególnych ramkach (łącznie z tą, którą projektujesz). Diagram powinien się skupiać na relacjach między Twoim urządzeniem, użytkownikami, serwerami, innymi urządzeniami i innymi jednostkami.

Jaki problem rozwiązuje system? Jakie są jego wejścia i wyjścia? Jakie są wejściowe i wyjściowe dane użytkowników? Jakie są wejściowe i wyjściowe dane szerszego systemu? Skup się na zastosowaniach i interakcjach ze światem zewnętrznym.

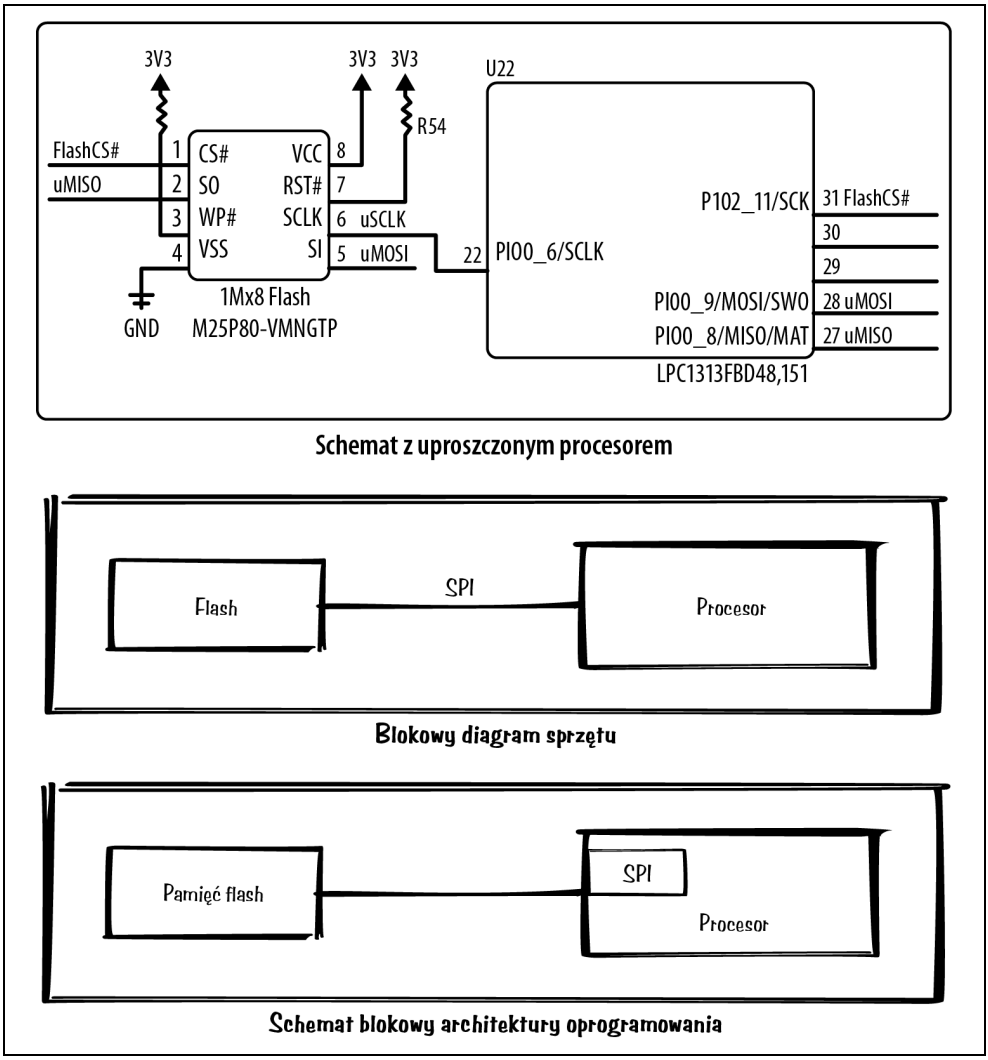
Ten typ diagramu może pomóc zdefiniować wymagania systemowe i przewidzieć prawdopodobne zmiany. W bardziej realistycznym scenariuszu pomaga pamiętać o celach urządzenia, w miarę jak zagłębiasz się w projektowanie oprogramowania.

Diagram blokowy

Projekt początkowo jest prosty, ponieważ rozważasz fizyczne elementy systemu i możesz myśleć w sposób obiektowy — bez względu na to, czy używasz języka obiektowego — aby modelować oprogramowanie wokół fizycznych elementów. Każdy układ scalony podłączony do procesora jest obiektem. Przewody łączące układ z procesorem (metody komunikacji) możesz traktować jak kolejny zbiór obiektów.

Zacznij projekt od narysowania tych obiektów jako prostokątów; proces znajduje się w środku rysunku, obiekty komunikacyjne — w procesorze, a każdy zewnętrzny komponent jest podłączony do obiektu komunikacyjnego.

Na potrzeby tego przykładu wprowadzę typ pamięci nazywany **pamięcią flash**. Szczegóły nie są tu istotne; jest to względnie niedrogi typ pamięci używany w wielu różnych urządzeniach. Wiele układów pamięci flash komunikuje się przez magistralę SPI (Serial Peripheral Interface, zwykle wymawiane „spaj"), typ łącza szeregowego (omówimy je dokładniej w rozdziale 7.). Większość procesorów nie może wykonywać kodu przez SPI, więc pamięć flash jest używana do przechowywania danych poza procesorem. Nasz schemat, widoczny na górze rysunku 2.2, pokazuje, że mamy pamięć flash podłączoną do procesora przez SPI. Nie bój się schematów! Opiszemy je szczegółowo w rozdziale 3.



Rysunek 2.2. Porównanie schematu oraz wstępnych diagramów sprzętu i oprogramowania

Byłoby idealnie, gdyby inżynier elektronik dostarczył Ci blokowy diagram sprzętu wraz ze schematem, aby zapewnić Ci uproszczony widok sprzętu. Jeśli nie, może będziesz musiał naszkicować go sam, kiedy będziesz pracować nad blokowym diagramem oprogramowania.

Na naszym blokowym diagramie oprogramowania dodamy pamięć flash jako urządzenie (prostokąt poza procesorem) oraz prostokąt SPI wewnątrz procesora, aby pokazać, że konieczne będzie napisanie kodu SPI. Na tym etapie nasz diagram oprogramowania jest bardzo podobny do diagramu sprzętu, ale w miarę identyfikowania dodatkowych komponentów programowych będzie coraz bardziej się różnić.

Następnym etapem będzie dodanie prostokąta „flash" wewnątrz procesora w celu zaznaczenia, że trzeba będzie napisać kod do obsługi pamięci flash. Warto oddzielić metodę komunikacji od zewnętrznego komponentu; jeśli jest wiele układów podłączonych z wykorzystaniem tej samej metody, wszystkie one powinny być połączone z tym samym blokiem komunikacyjnym w procesorze. W ten sposób diagram ostrzeże nas, że powinniśmy zachować szczególną ostrożność odnośnie do współdzielonego zasobu oraz rozważyć kwestie wydajności i ewentualnych konfliktów, które wiążą się ze współdzieleniem.

Na rysunku 2.2 pokazano fragment schematu oraz początki blokowego diagramu oprogramowania. Zauważ, że schemat jest znacznie bardziej szczegółowy. Na tym etapie projektowania chcemy zobaczyć elementy ogólne, aby ustalić, jakie obiekty będziemy musieli utworzyć i jak je ze sobą połączymy. Szczegóły trzymaj w pamięci (albo na oddzielnej kartce papieru), zwłaszcza jeśli mogą one mieć wpływ na system, ale w razie możliwości nie nanoś ich na diagram. Celem jest rozłożenie systemu na małe kawałki, które można podsumować prostokątem na diagramie.

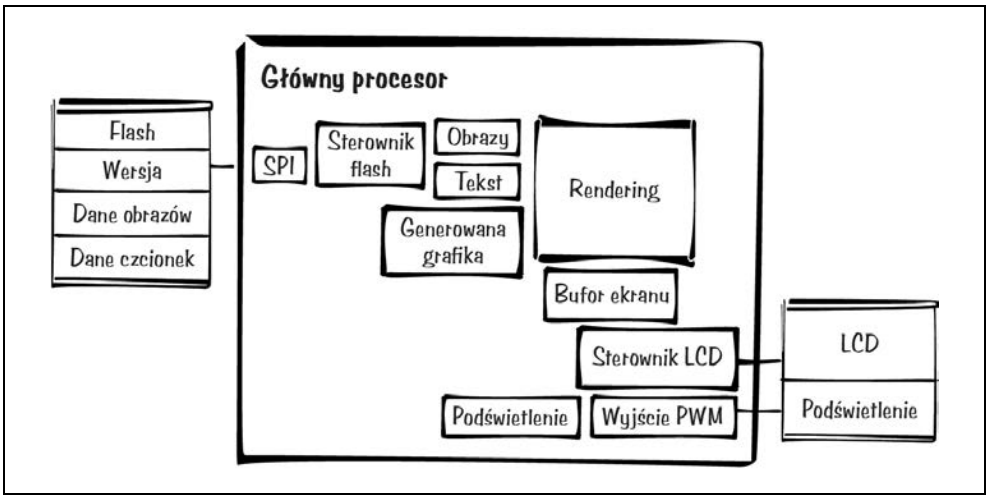
Następnym etapem jest dodanie funkcji wyższego poziomu. Do czego służy każdy zewnętrzny układ? Jest to proste, jeśli każdy ma tylko jedną funkcję. Jeśli na przykład pamięć flash służy do przechowywania bitmap, które będą wyświetlane na ekranie, możemy dodać do architektury prostokąt reprezentujący zasoby graficzne. Nie są one związane z żadnym elementem spoza procesora, więc ich prostokąt umieszczamy wewnątrz procesora. Będziemy też potrzebować prostokąta dla ekranu i jego metody komunikacyjnej oraz kolejnego prostokąta odpowiadającego za przekazywanie zasobów graficznych z pamięci flash na ekran. Na tym etapie lepiej mieć za dużo prostokątów niż za mało. Później będzie można je połączyć.

Dodaj inne struktury programowe, które przychodzą Ci do głowy: bazy danych, systemy buforowania, procedury obsługi poleceń, algorytmy, maszyny stanów itd. Możesz nie wiedzieć, co będzie potrzebne, żeby je zrealizować (porozmawiamy o tym dalej w tej książce), ale spróbuj przedstawić na diagramie wszystko od sprzętu do funkcji produktu (rysunek 2.3).

Po naszkicowaniu tego diagramu na kartce papieru albo na tablicy (prawdopodobnie wiele razy, bo prostokąty zawsze okazują się za małe albo znajdują się w niewłaściwym miejscu) możesz pomyśleć, że na tym koniec. Jednak inne typy diagramów mogą dostarczyć Ci dodatkowych informacji.

Jeśli próbujesz zrozumieć istniejący kod, sporządź listę plików z kodem. Jeśli jest ich zbyt wiele, użyj nazw katalogów i bibliotek. Ktoś zdecydował, że będą to moduły, a może nawet spróbował potraktować je jako obiekty, więc trafiają one na nasz diagram jako prostokąty.

To, gdzie umieścić prostokąty i jak je uporządkować, jest łamigłówką, której rozwiązanie może zająć trochę czasu.



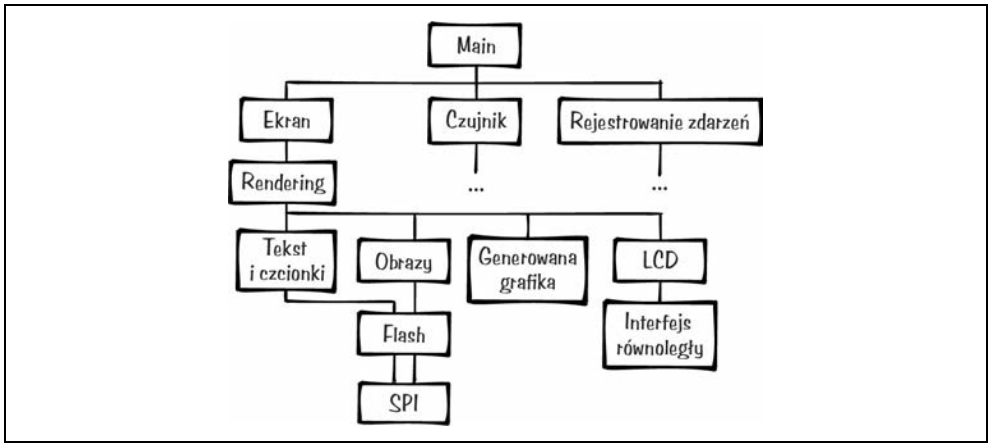
Rysunek 2.3. Bardziej szczegółowy blokowy diagram oprogramowania

Inne typy rysunków mogą ukazać ukryte, brzydkie miejsca z krytycznymi wąskimi gardłami, źle zrozumiane wymagania albo niemożność zaimplementowania funkcji produktu na zamierzonej platformie. Wady te są często widoczne tylko na jednym rysunku albo są reprezentowane przez prostokąty, które zmieniają się najbardziej między różnymi diagramami. Patrząc na nie z odpowiedniej perspektywy, przy odrobinie szczęścia będziesz mógł zidentyfikować problematyczne moduły i znaleźć drogę do dobrego rozwiązania.

Organigram

Kolejny typ diagramu architektury oprogramowania, **organigram**, wygląda jak schemat organizacyjny. W rzeczywistości chciałabym, żebyś myślał o nim jak o schemacie organizacyjnym. Podobnie jak kierownicy, komponenty wyższego szczebla mówią tym położonym niżej, co mają robić. Komunikacja nie jest (nie powinna być!) jednokierunkowa. Elementy niższego poziomu robią, co w ich mocy, aby wykonać polecenie, dostarczają żądanych informacji i powiadamiają szefa o ewentualnych błędach. Pomyśl o systemie jak o hierarchii, a ten diagram pokaże kontrolę i zależności.

Na rysunku 2.4 pokazano dyskretne komponenty oraz to, które z nich wywołują inne. Cały system jest hierarchią z procedurą `main` na najwyższym poziomie. Jeśli zaplanowałeś algorytm urządzenia i wiesz, jak będzie on używać każdego elementu, możesz wypełnić następny poziom obiektami związanymi z algorytmem. Jeśli nie sądzisz, żeby miały się one zmienić, możesz zacząć do funkcji związanych z produktem, a potem przejść niżej do elementów, które już znasz, umieszczając najbardziej złożone na samej górze. Następnie dodaj obiekty niższego poziomu, które są używane przez obiekt wyższego poziomu. Na przykład nasz obiekt SPI jest używany przez obiekt flash, który jest używany przez obiekt zasobów graficznych itd. Możesz tu dodać pewne elementy, o których wcześniej nie pomyślałeś. Powinieneś też (prawdopodobnie) ustalić, gdzie umieścić te elementy na diagramie blokowym.



Rysunek 2.4. Diagram organizacyjny architektury oprogramowania

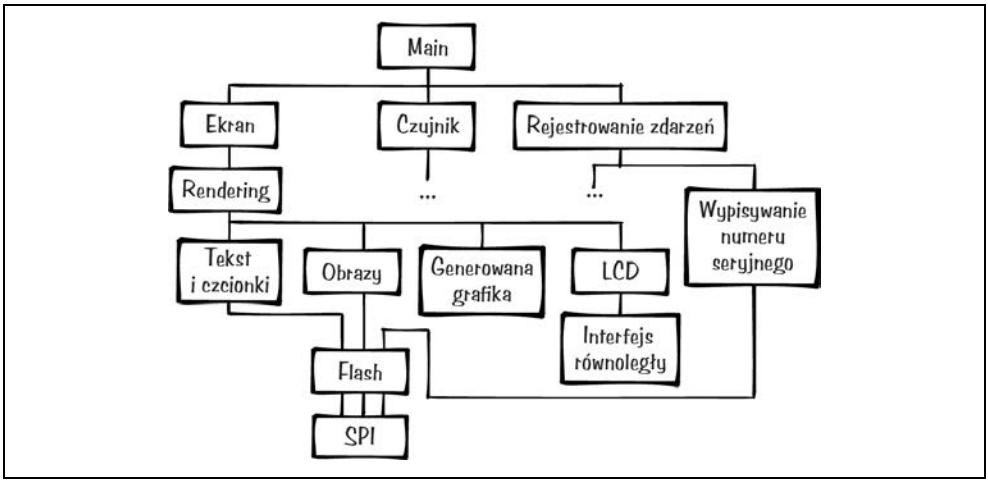
Choćbyśmy jednak chcieli przypisać jeden komponent do jednej funkcji (np. zasoby graficzne w pamięci flash), ograniczenia systemu (koszt, szybkość itd.) nie zawsze na to pozwalają. Często ostatecznie musisz upchnąć wiele nieszczególnie kompatybilnych funkcji w jednym komponencie albo w ścieżce komunikacyjnej.

Na diagramie widać, jak tekst i obrazy współdzielą sterownik flash i jego podrzędny sterownik SPI. Takie współdzielenie bywa konieczne, ale jest czerwoną flagą w projekcie, bo trzeba będzie zachować szczególną ostrożność, aby uniknąć współzawodnictwa o zasoby i upewnić się, że będą dostępne, kiedy urządzenie będzie ich potrzebować. Na szczęście rysunek pokazuje, że kod renderingu kontroluje oba moduły i będzie mógł zagwarantować, że w danym momencie będzie potrzebny tylko jeden z zasobów — tekst lub obrazy — więc konflikt między nimi jest mało prawdopodobny.

Przypuśćmy, że Twój zespół ustalił, że w projektowanym systemie każda jednostka będzie miała numer seryjny. Ma on być programowany podczas produkcji i przekazywany na żądanie. Moglibyśmy dodać kolejny układ pamięciowy jako komponent, ale to zwiększyłoby koszty, złożoność płytki i złożoność oprogramowania. Pamięć flash, którą już mamy, jest wystarczająco duża, aby pomieścić numer seryjny. Dzięki temu zwiększa się tylko złożoność oprogramowania. (Ech!)

Na rysunku 2.5 wypisujemy numer seryjny systemu z wykorzystaniem pamięci flash, która wcześniej była przeznaczona tylko na zasoby graficzne. Jeśli podsystem rejestrowania zdarzeń musi pobierać numer seryjny asynchronicznie względem podsystemu zasobów graficznych (przypuśćmy, że mamy dwa wątki albo zasoby graficzne są używane w przerwaniu), oprogramowanie będzie musiało unikać kolizji i spowodowanych nimi uszkodzeń danych.

Za każdym razem, kiedy dodajesz jakiś mały element, tak że używasz A oraz B i musisz rozważyć potencjalne interakcje z C, system staje się nieco mniej solidny. Trudno to udokumentować, a współdzielone zasoby powodują problemy w fazie projektowania, implementowania i konserwowania produktu. Powyższy przykład można łatwo naprawić przez dodanie flagi, ale wszystkie współdzielone zasoby powinny skłaniać Cię do myślenia o potencjalnych konsekwencjach.



Rysunek 2.5. Diagram organizacyjny ze współdzielonym zasobem

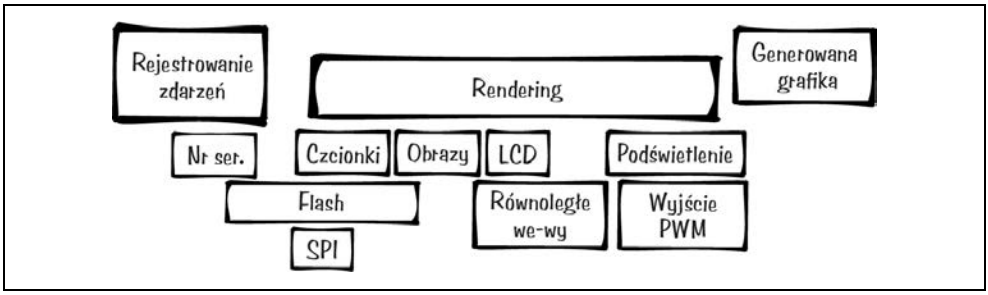
Wracając do metafory schematu organizacyjnego, osoby mające dwóch przełożonych często będą otrzymywać sprzeczne instrukcje. Choć najlepiej jest unikać takiej sytuacji, czasem musisz zarządzać sobą sam, żeby poradzić sobie z zawiłościami pracy.

Jeśli próbujesz zrozumieć istniejący kod, możesz skonstruować organigram poprzez krokowe wykonywanie kodu w debugerze. Zaczynasz w `main()` i wchodzisz w interesujące funkcje, starając się nie zgubić w detalach.

Może to wymagać kilku powtórzeń, ale celem jest zorientowanie się w ogólnym przepływie kodu.

Diagram warstwowy

Ostatni rysunek architektoniczny pokazuje warstwy oprogramowania i reprezentuje obiekty według ich szacowanego rozmiaru, jak pokazano na rysunku 2.6. To kolejny diagram, który warto najpierw narysować ołówkiem na papierze. Zacznij od dołu strony i narysuj prostokąty dla elementów, które łączą procesor ze światem zewnętrznym (takich jak moduły komunikacyjne). Jeśli przewidujesz, że implementacja któregoś z nich może być bardziej skomplikowana, nieco go powiększ. Jeśli nie jesteś pewien, narysuj wszystkie w tym samym rozmiarze.



Rysunek 2.6. Diagram warstwowy architektury oprogramowania

Następnie dodaj do diagramu elementy, które używają najniższej warstwy. Jeśli obiekt niższego poziomu ma wielu użytkowników, wszyscy oni powinni dotykać tego obiektu (może to wymagać powiększenia komponentu niższego poziomu). Ponadto każdy obiekt, który używa czegoś pod spodem, powinien dotykać wszystkich używanych obiektów, jeśli to możliwe.

Było to trochę podstępne. Powiedziałam, że rozmiar obiektu zależy od jego złożoności. Potem powiedziałam, że jeśli obiekt ma wielu użytkowników, to należy go powiększyć. Jak opisano w poprzednim punkcie, współdzielone zasoby zwiększają złożoność. Kiedy więc zasób jest współdzielony przez wiele rzeczy, jego prostokąt rośnie, żeby mógł dotknąć wszystkich wyższych modułów. Odzwierciedla to wzrost złożoności, nawet jeśli moduł wydaje się prosty. Problem ten pojawia się nie tylko w przypadku dolnych warstw. Na moim diagramie prostokąt renderingu początkowo był dużo mniejszy, ponieważ przenoszenie danych z pamięci flash na ekran LCD jest łatwe. Kiedy jednak okazało się, że moduł renderingu musi kontrolować różne znajdujące się niżej elementy, jego prostokąt stał się większy. I rzeczywiście, w projekcie, z którego pochodzi ten diagram, ostatecznie rendering stał się dość dużym modułem, a wreszcie dwoma modułami.

Ostatecznie diagram warstwowy pokazuje, jakie warstwy znajdują się w Twoim kodzie, co pozwala zgrupować zasoby, jeśli są one zawsze używane razem. Na przykład prostokąty LCD i równoległego wejścia-wyjścia dotykają tylko siebie nawzajem. Jeśli to jest ostateczny diagram, może można je połączyć w jeden moduł. To samo dotyczy podświetlenia i wyjścia PWM.

Przyjrzyj się też zgrupowaniom poziomym. Czcionki i obrazy dzielą połączenia wyższego i niższego poziomu. Możliwe, że należałoby połączyć je w jeden moduł, bo wydaje się, że mają te same wejścia i wyjścia. Diagram warstwowy pomaga wyszukać takie punkty i przemyśleć implikacje łączenia prostokątów na różne sposoby. Dzięki temu można uzyskać prostszy projekt.

Wreszcie, jeśli masz grupę kilku modułów, które dotykają pewnego elementu niższego poziomu, możesz się zastanowić nad podzieleniem tego elementu. Czy mógłby się przydać sterownik flash, który zajmowałby się tylko numerem seryjnym? Może wczytywałby on numer seryjny podczas inicjalizacji i nigdy nie robił tego ponownie, żeby podsystem ekranu stale miał kontrolę nad pamięcią flash? Postaraj się zrozumieć złożoność swojego projektu i opcje projektowania systemu, które pozwolą zapanować nad tą złożonością. Dobry projekt może oszczędzić czas i pieniądze w fazach implementacji i konserwacji.

Projektowanie pod kątem zmian

Teraz, kiedy siedzisz nad stosem rysunków architektonicznych, powstaje pytanie: co dalej? Może uświadomiłeś sobie, że jest kilka fragmentów kodu, o których początkowo nie pomyślałeś. Może wiesz już nieco więcej o interakcjach między modułami. Zanim rozważymy te interakcje (interfejsy), warto spędzić trochę czasu nad jednym ważnym pytaniem: *co będzie się zmieniać?* Na tym etapie wszystko ma charakter eksperymentalny, więc jest duża szansa, że może się zmienić każdy kawałek systemowej układanki. Celem jest jednak przygotowanie wstępnej architektury (i kodu) na możliwe zmiany funkcji systemu lub rzeczywistego sprzętu.

Z wymagań produktowych może wynikać, że niektóre funkcje systemu się nie zmienią. Nasze przykładowe urządzenie, bez względu na to, co robi, potrzebuje ekranu, a najlepszym sposobem przesyłania bitmap na ekran wydaje się pamięć flash. Wiele układów flash obsługuje SPI, więc to również wydaje się rozsądnym założeniem. Jednak konkretny układ flash prawdopodobnie się zmieni. Wyświetlacz LCD, obrazy i dane czcionek także mogą się zmienić. Zmienić się może nawet sposób, w jaki przechowujesz obrazy lub dane czcionek. Prostokąty na ekranie powinny reprezentować platoniczne ideały, a nie konkretne reprezentacje.

Enkapsulacja modułów

Proces tworzenia diagramów prowadzi do interfejsów, które nie zależą od konkretnej zawartości lub zachowania łączonych modułów (to właśnie jest enkapsulacja!). Używamy różnych rysunków architektonicznych, aby znaleźć najlepsze miejsca na te interfejsy. Każdy prostokąt prawdopodobnie będzie miał swój własny interfejs. Może niektóre można połączyć w jeden obiekt. Ale czy jest dobry powód, żeby zrobić to teraz, a nie później?

Czasem tak. Jeśli możesz ograniczyć złożoność swoich diagramów bez poświęcania przyszłej elastyczności, prawdopodobnie warto przerzedzić niektóre drzewa zależności.

Oto kilka rzeczy, których należy szukać:

- Na organigramie poszukaj obiektów, które są używane przez tylko jeden inny obiekt. Czy są one niezmienne? Jeśli prawdopodobnie nie będą się zmieniać albo będą się zmieniać razem w miarę ewolucji systemu, rozważ ich połączenie. Jeśli zmieniają się niezależnie, nie są dobrymi kandydatami do enkapsulacji.
- Na diagramie warstwowym poszukaj zbiorów obiektów, które są zawsze używane razem. Czy można połączyć je w interfejsie wyższego poziomu? W ten sposób utworzyłbyś warstwę abstrakcji sprzętowej.
- Które moduły mają dużo wzajemnych zależności? Czy można je oddzielić i uprościć? Czy lepiej zgrupować zależności?
- Czy interfejsy między pionowymi sąsiadami można opisać kilkoma zdaniami? Jeśli tak, jest to dobre miejsce na enkapsulację w celu utworzenia interfejsu (co pozwoli innym ponownie wykorzystać Twój kod albo po prostu ułatwi testy).

W przykładowym systemie wyświetlacz LCD jest podłączony do interfejsu równoległego. Na każdym diagramie jest prosty podsystem, bez dodatkowych zależności i bez innych podsystemów wymagających dostępu do interfejsu równoległego. Nie musisz eksponować interfejsu równoległego; możesz enkapsulować go (ukryć) w module LCD.

Odwrotnie, wyobraź sobie system bez modułu renderingu, który enkapsuluje podsystem ekranu. Najlepiej widać to na diagramie warstwowym (rysunek 2.5). Czcionki, obrazy, LCD i podświetlenie próbowałyby dotknąć siebie nawzajem, tworząc bałagan na diagramie.

Każdy prostokąt, który pozostanie na diagramie, prawdopodobnie stanie się modułem (lub obiektem). Przyjrzyj się rysunkowi. Jak możesz go uprościć? Enkapsulacja w oprogramowaniu przekłada się na bardziej uporządkowany rysunek architektoniczny; działa to również w drugą stronę.

Delegowanie zadań

Diagram pomaga też podzielić pracę i przydzielić ją podwykonawcom. Które części systemu mogą się stać oddzielnymi kawałkami, które zaimplementuje ktoś inny?

Zbyt często chcemy, żeby podwykonawcom przypadły nudne części pracy, podczas gdy my zajmiemy się tym, co interesujące („Proszę, podwykonawco, napisz mój kod testowy, zrób dokumentację, poskładaj pranie"). Nie tylko odstrasza to dobrych podwykonawców, ale również zmniejsza jakość Twojego produktu. Zamiast tego pomyśl, które prostokąty (albo całe poddrzewa) możesz oddać w inne ręce. Próbując opisać wzajemne zależności wyimaginowanemu podwykonawcy, możesz zauważyć, że są one gorsze, niż sugerowałyby diagramy. Możesz też odkryć, że prosta flaga (taka jak semafor), która opisuje, do kogo obecnie należy zasób, jest wystarczająco dobrym rozwiązaniem.

A jeśli nie masz widoków na pozyskanie podwykonawców? Nadal warto przejść przez ten proces myślowy. Powinieneś ograniczać wzajemne zależności wszędzie, gdzie to możliwe, co może skłonić Cię do przeprojektowania systemu. A jeśli nie da się ich ograniczyć, powinieneś przynajmniej pamiętać o nich podczas kodowania.

Szukanie rzeczy, które można oddzielić i powierzyć innej osobie, pomoże Ci zidentyfikować sekcje kodu, między którymi jest prosty interfejs. Ponadto, kiedy dział marketingu zapyta, jak można przyspieszyć projekt, będziesz mieć gotową odpowiedź. Wyobrażony podwykonawca zapewnia jednak jeszcze jedną korzyść: załóż, że jest on trochę nierozgarnięty i że musisz chronić siebie oraz swój kod przed jego złym kodem.

Jakie rodzaje struktur defensywnych możesz wznieść między modułami? Wyobraź sobie dane przekazywane między modułami. Jaka jest minimalna ilość danych, które mogą przepływać między prostokątami (albo grupami prostokątów)? Czy dodanie prostokąta do grupy spowoduje znaczne zmniejszenie ilości przekazywanych danych? Jak przechowywać dane w sposób, który zapewni im bezpieczeństwo i pozwoli używać ich wszystkim, którzy ich potrzebują?

Minimalizacja złożoności między prostokątami (a przynajmniej między grupami prostokątów) sprawi, że projekt będzie przebiegał sprawniej. Im bardziej Twoi podwykonawcy są ograniczeni do własnych podzbiorów kodu, z dobrze zdefiniowanymi interfejsami, tym łatwiej każdemu testować i rozwijać własny kod.

Interfejs sterownika: open, close, read, write, ioctl

W poprzednim podrozdziale przedstawiłam interfejsy modułów w widoku „od góry w dół", żeby zachęcić Cię do rozważenia enkapsulacji i myślenia o tym, gdzie możesz uzyskać pomoc od innych osób. Można też zacząć od dołu i poruszać się w górę. „Dół" składa się tu z modułów, które komunikują się ze sprzętem (sterowników).

Projektowanie „od góry w dół" ma miejsce wtedy, gdy myślisz o tym, czego chcesz, a następnie zastanawiasz się nad tym, czego będziesz potrzebował do osiągnięcia celu. Projektowanie „od dołu w górę" ma miejsce wtedy, kiedy rozważasz to, co masz do dyspozycji, i próbujesz coś z tego zbudować.

Ja zwykle stosuję kombinację obu podejść: *projektowanie yo-yo*.

Wiele sterowników w systemach wbudowanych jest opartych na interfejsie POSIX API używanym do wywoływania urządzeń w systemach uniksowych. Dlaczego? Ponieważ model ten dobrze sprawdza się w wielu sytuacjach i oszczędza Ci ponownego wymyślania koła za każdym razem, kiedy potrzebujesz dostępu do sprzętu.

Interfejs sterowników uniksowych jest prosty:

`open`
: Otwiera sterownik do użytku. Podobnym (i często stosowanym zamiennie) wywołaniem jest `init`.

`close`
: Zamyka sterownik po użyciu.

`read`
: Wczytuje dane z urządzenia.

`write`
: Wysyła dane do urządzenia.

`ioctl` (wym. „ajoktal” albo „aj ou kontrol”)
: Oznacza sterowanie wejściem-wyjściem (I/O) i obsługuje funkcje, które nie są dostępne za pośrednictwem innych części interfejsu. Programiści jądra trochę krzywią się na użycie tego wywołania ze względu na brak struktury, ale jest ono wciąż bardzo popularne.

W Uniksie sterownik jest częścią jądra. Każda z tych funkcji przyjmuje argument w postaci deskryptora pliku, który reprezentuje dany sterownik (np. */dev/tty01* w przypadku pierwszego terminala w systemie). Byłoby to kłopotliwe w systemie wbudowanym, który nie ma systemu operacyjnego. Przykładowa funkcja w urządzeniu wbudowanym mogłaby wyglądać tak jak dowolna z poniższych[1]:

- `spi.open()`,
- `spi_open()`,
- `SpiOpen(WITH_LOCK)`,
- `spi.ioctl_changeFrequency(THIRTY_MHz)`,
- `SpiIoctl(kChangeFrequency, THIRTY_MHz)`.

Interfejs ten ukrywa szczegóły sterownika, przez co jest on mniej specyficzny dla danego zastosowania, a kod nadaje się do wielokrotnego użytku. Co więcej, kiedy przekażesz swój kod komuś innemu, a sterownik będzie obejmował te funkcje, Twój następca będzie wiedział, czego się spodziewać.

[1] Styl jest bardzo ważny. Wytyczne dotyczące kodowania oszczędzą Ci sporo debugowania i nie ograniczą Twojej kreatywności. Jeśli nie masz przewodnika po stylu, przyjrzyj się tym, które Google zaleca dla projektów open source (wyszukaj „Google Style Guides”). Wyjaśnienia wybranych konwencji pomogą Ci sformułować własny przewodnik.

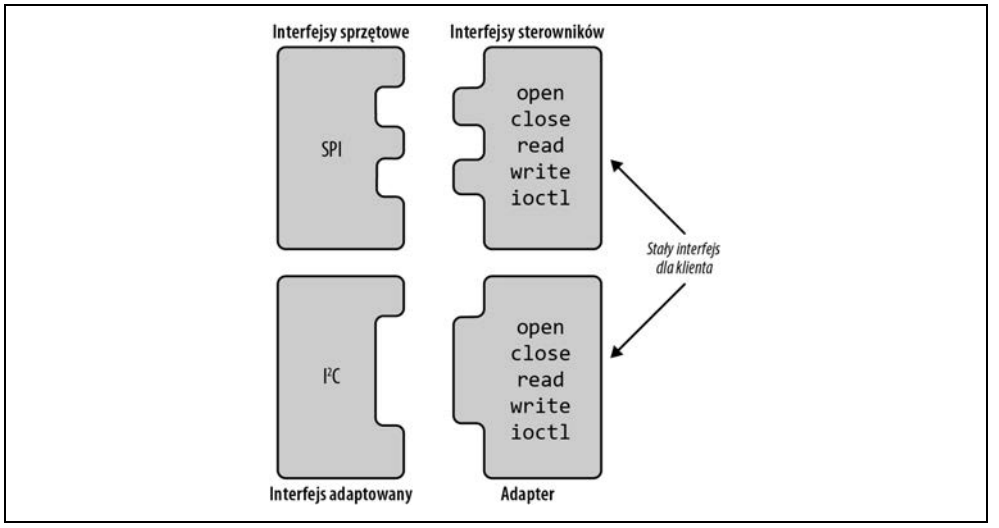
Model sterowników w Uniksie czasem obejmuje dwie nowsze funkcje. Pierwsza, `select` (lub `poll`), czeka, aż urządzenie zmieni stan. Wcześniej wykorzystywano do tego puste odczyty albo komunikaty `ioctl`, ale obecnie operacja ta ma własną funkcję. Druga funkcja, `mmap`, kontroluje mapę pamięci, którą sterownik współdzieli z wywołującym go kodem.

Jeśli Twój okrągły kołek nie pasuje do kwadratowego otworu zgodnego ze standardem POSIX, nie wpychaj go na siłę. Ale jeśli wygląda na to, że może się to udać, zaczęcie od tego standardowego interfejsu może sprawić, że Twój projekt będzie nieco lepszy i łatwiejszy w konserwacji.

Wzorzec adaptera

Jednym z tradycyjnych wzorców projektowania oprogramowania jest **adapter** (czasem nazywany **nakładką**). Przekształca on interfejs obiektu tak, żeby był łatwiejszy w użyciu dla klienta (modułu wyższego poziomu). Adaptery często stosuje się w połączeniu z programowymi interfejsami API w celu ukrycia brzydkich interfejsów albo bibliotek, które się zmieniają.

Wiele interfejsów sprzętowych przypomina nieporęczne interfejsy programowe. Dlatego każdy sterownik jest adapterem, jak pokazano na rysunku 2.7. Jeśli utworzysz standardowy interfejs sterownika (nawet jeśli nie będą to wywołania `open`, `close`, `read`, `write`, `ioctl`), interfejs sprzętowy będzie mógł się zmienić bez modyfikowania oprogramowania wyższego poziomu. W najlepszym przypadku będziesz mógł zmienić całą platformę sprzętową i przerobić tylko kod najniższego poziomu.



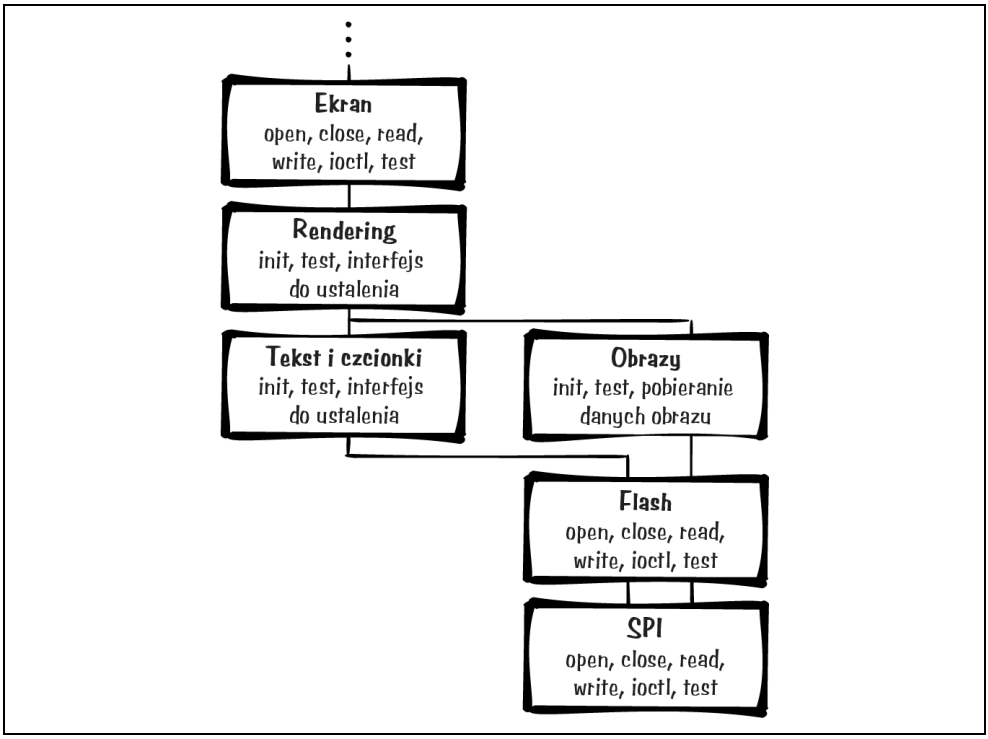
Rysunek 2.7. Sterownik implementuje wzorzec adaptera

Zauważ, że sterowniki można układać jeden na drugim. W naszym przykładzie mamy ekran, który używa pamięci flash, która z kolei używa komunikacji SPI. Kiedy wywołujesz funkcję `open` ekranu, wywołuje ona kod inicjalizujący podsystemy ekranu, który wywołuje funkcję `open` pamięci flash, ta zaś wywołuje funkcję `open` sterownika SPI. Są to trzy poziomy adapterów, a wszystko to dla przenośności i łatwości konserwacji.

Warstwy i adaptery zwiększają złożoność implementacji. Mogą też wymagać więcej pamięci albo wprowadzać opóźnienia w kodzie. Jest to kompromis. Za niewielką cenę zyskujesz prostszą architekturę, którą łatwiej się konserwuje, testuje, przenosi na inne platformy itd. Jeśli Twój system nie jest bardzo ograniczony pod względem zasobów, zwykle jest to dobry interes.

Jeśli interfejs na każdym poziomie pozostaje niezmienny, kod wyższego poziomu jest dość odporny na zmiany. Jeśli na przykład zmienimy pamięć SPI flash na pamięć I^2C EEPROM (inny typ pamięci z inną magistralą komunikacyjną), może nie trzeba będzie zmieniać sterownika ekranu albo konieczne będzie tylko zastąpienie funkcji flash funkcjami obsługującymi pamięć EEPROM (kasowalną elektronicznie programowalną pamięć przeznaczoną tylko do odczytu).

Na rysunku 2.8 dodałam funkcję o nazwie `test` do interfejsów każdego modułu. W rozdziale 3. omówię strategie automatycznego testowania, które mogą zwiększyć niezawodność Twojego kodu. Na razie po prostu rezerwuję miejsce na te funkcje.



Rysunek 2.8. Interfejsy podsystemu ekranu i modułów podrzędnych

Tworzenie interfejsów

Definicja interfejsu modułu zależy od specyfiki systemu. Można bezpiecznie założyć, że większość modułów będzie też potrzebowała funkcji inicjalizacyjnej (choć sterowniki często używają do tego wywołania `open`). Inicjalizacja może się odbywać w miarę tworzenia obiektów

podczas rozruchu urządzenia albo może być funkcją wywoływaną w czasie inicjalizacji systemu. Aby zapewnić enkapsulację modułów (i ułatwić ich wielokrotne wykorzystywanie), funkcje wysokiego poziomu powinny być odpowiedzialne za inicjalizowanie modułów, od których są zależne. Dobrą funkcję `init` można wywoływać wiele razy, jeśli jest używana przez różne podsystemy. Bardzo dobra funkcja `init` może zresetować podsystem (lub zasób sprzętowy) do znanego dobrego stanu w przypadku częściowej awarii systemu.

Teraz, kiedy nie masz już zupełnie czystej karty, zapewne będzie Ci łatwiej dodać interfejs do każdego prostokąta. Zastanów się, jak zachować enkapsulację modułu, wkładu Twojego hipotetycznego podwykonawcy oraz modelu sterownika, a następnie zacznij określać obowiązki każdego prostokąta na swoich diagramach architektonicznych.

Utworzywszy różne widoki architektury, prawdopodobnie nie będziesz chciał aktualizować każdego z nich. W miarę dodawania interfejsów skup się na tym diagramie, który wydaje Ci się najbardziej użyteczny (albo jest najbardziej klarowny dla Twojego szefa lub podwykonawców).

Przykład: interfejs rejestrowania zdarzeń

Celem interfejsu rejestrowania zdarzeń jest zaimplementowanie niezawodnego, nadającego się do wielokrotnego użytku systemu rejestrowania zdarzeń. W tym punkcie zaczniemy od zdefiniowania wymagań wobec interfejsu, a następnie zbadamy różne opcje implementacji interfejsu (i lokalnej pamięci). Metoda komunikacji nie ma znaczenia. Kodując interfejs z myślą o ograniczeniach, otwierasz sobie furtkę do ponownego wykorzystania kodu w innym systemie.

Rejestrowanie komunikatów diagnostycznych może znacznie spowalniać procesor. Jeśli działanie Twojego kodu zmienia się, kiedy włączasz i wyłączasz rejestrowanie, zastanów się nad zależnościami czasowymi pomiędzy różnymi podsystemami.

Implementacja zależy od systemu. Czasem najlepsze, co możesz zrobić, to przełączać stan linii wejścia-wyjścia podłączonej do diody LED i wysyłać komunikaty alfabetem Morse'a (wcale nie żartuję). Jednak w większości przypadków będziesz mógł wypisywać tekstowe komunikaty diagnostyczne do jakiegoś interfejsu. System, który można debugować, to system łatwiejszy w konserwacji. Nawet jeśli Twój kod jest idealny, osoba, która przyjdzie po Tobie, może nie mieć tyle szczęścia, kiedy doda nowo wymaganą funkcję. Podsystem rejestrowania zdarzeń przyda się nie tylko podczas tworzenia urządzenia, ale będzie również nieocenioną pomocą w trakcie konserwacji.

Dobry interfejs rejestrowana zdarzeń zasłania szczegóły rzeczywistego rejestrowania, pozwalając Ci ukryć zmiany (i złożoność). W cyklu projektowym potrzeby w zakresie rejestrowania zdarzeń mogą się zmieniać. Na przykład we wstępnej fazie Twoja płytka rozwojowa może mieć dodatkowy port szeregowy, przez który można przesyłać informacje diagnostyczne do komputera. W późniejszej fazie port szeregowy może być niedostępny, więc niewykluczone, że będziesz musiał ograniczyć informacje diagnostyczne do jednej lub dwóch diod LED.

Czasem, kiedy system zachowywał się dziwnie, żałowałam, że nie mogę połączyć się z nim telepatycznie. Niestety, nigdy nie ma dość zasobów, żeby zgromadzić wszystkie informacje, których byśmy sobie życzyli. Co więcej, metodyka logowania może się zmieniać w miarę rozwoju produktu. Jest to obszar, w którym powinieneś zadbać o enkapsulację zmian poprzez ukrycie funkcji wywoływanych z głównego interfejsu. Stanowi to niewielkie dodatkowe obciążenie, ale zwiększa elastyczność. Jeśli możesz kodować pod kątem interfejsu, zmiana zakulisowych operacji nie będzie miała znaczenia.

Często będziesz chciał przesyłać mnóstwo informacji przez względnie niewielkie łącze. W miarę powiększania się systemu łącze to będzie się wydawać coraz mniejsze. Może nim być prosty port szeregowy łączący urządzenie z komputerem i dostępny tylko w specjalnej wersji sprzętu. Może to być specjalny pakiet diagnostyczny przesyłany przez sieć. Dane mogą być przechowywane w zewnętrznej pamięci RAM i dostępne tylko wtedy, gdy wstrzymasz procesor i odczytasz je przez JTAG. Dziennik zdarzeń może być dostępny tylko wtedy, gdy wykonujesz kod na płytce rozwojowej, ale nie na samodzielnie zaprojektowanym sprzęcie. Zatem pierwsze ważne wymaganie wobec tego modułu jest następujące: interfejs rejestrowania zdarzeń powinien obsługiwać różne podstawowe implementacje.

Po drugie, ponieważ w danym momencie pracujesz tylko nad jednym obszarem systemu, możesz nie potrzebować (albo nie chcieć) komunikatów z innych obszarów. Wymaganie jest zatem takie, że metody rejestrowania powinny być specyficzne dla podsystemu. Oczywiście, powinieneś wiedzieć o katastrofalnych awariach w innych podsystemach.

Trzecim wymaganiem jest poziom priorytetu pozwalający Ci debugować szczegóły podsystemu, nad którym właśnie pracujesz, bez utraty krytycznych informacji z innych części kodu.

Od wymagań do interfejsu

Zdefiniowanie podstawowych wymagań wobec interfejsu modułu często wystarcza, zwłaszcza w fazie projektowej. To jednak sporo słów jak na coś, co można podsumować jednym wierszem kodu:

```
void Log(enum eLogSubSystem sys, enum eLogLevel level, char *msg);
```

Ten prototyp nie jest wyryty w kamieniu i może się zmieniać w miarę rozwoju interfejsu, ale zapewnia przydatny szablon dla innych programistów uczestniczących w projekcie.

Poziomy rejestrowania mogą obejmować brak rejestrowania, komunikaty informacyjne, komunikaty diagnostyczne, ostrzeżenia, błędy i błędy krytyczne. Podsystemy będą zależeć od Twojego systemu, ale mogą obejmować komunikację, ekran, czujniki, aktualizację oprogramowania układowego itd.

Zauważ, że funkcja `Log` przyjmuje łańcuch, inaczej niż `printf` lub `iostream`, które przyjmują zmienną liczbę argumentów. W razie potrzeby zawsze możesz użyć biblioteki, aby uzyskać podobne formatowanie, a nawet obsługę zmiennej liczby argumentów. Jednak rodziny funkcji `printf` i `iostream` to jedne z pierwszych rzeczy, które usuwa się z systemu potrzebującego więcej przestrzeni kodu i pamięci. W takim przypadku prawdopodobnie będziesz musiał sam zaimplementować potrzebne Ci funkcje, więc interfejs rejestrowania powinien reprezentować

niezbędne minimum. Oprócz wypisywania łańcuchów zwykle konieczne będzie przynajmniej wypisywanie po jednej liczbie naraz:

```
void LogWithNum(enum eLogSubSystem sys, enum eLogLevel level, char *msg, int number);
```

Użycie identyfikatorów podsystemów i poziomów priorytetu umożliwia zdalne zmienianie opcji debugowania (jeśli Twój system pozwala na takie rzeczy). Kiedy coś debugujesz, możesz ustawić niski poziom priorytetu (np. komunikaty diagnostyczne), a po pomyślnym przetestowaniu podsystemu możesz zwiększyć jego poziom priorytetu (np. błędy). W ten sposób będziesz otrzymywał dokładnie te komunikaty, których potrzebujesz, wtedy, kiedy ich potrzebujesz. Interfejs musi zatem zapewniać tę elastyczność wyboru podsystemu i poziomu priorytetu:

```
void LogSetOutputLevel(enum eLogSubSystem sys, enum eLogLevel level)
```

Ponieważ wywołujący kod nie dba o rzeczywistą implementację, nie powinien mieć do niej bezpośredniego dostępu. Wszystkie operacje rejestrowania powinny przechodzić przez ten interfejs. Byłoby najlepiej, gdyby podstawowy interfejs nie był współdzielony z żadnym innym modułem, ale jeśli jest inaczej, dowiesz się tego ze swoich diagramów architektonicznych. Funkcja inicjalizacji rejestrowania powinna wywoływać moduł, od którego jest zależna, czy będzie to inicjalizacja sterownika portu szeregowego, czy linii wejścia-wyjścia.

Ponieważ rejestrowanie może zmieniać zależności czasowe w systemie, czasem system rejestrowania trzeba wyłączyć globalnie. Pozwala to potwierdzić z dużą dozą pewności, że podsystem diagnostyczny nie zakłóca działania innych części kodu. Choć taki wyłącznik zapewne nie będzie używany często, to stanowi doskonały dodatek do interfejsu:

```
void LogGlobalOn();
void LogGlobalOff();
```

Kontrola wersji kodu

W pewnym momencie ktoś będzie musiał wiedzieć, jaka dokładnie wersja kodu działa w Twoim urządzeniu. W świecie aplikacji komputerowych umieszczenie numeru wersji w oknie pomocy jest prostą sprawą. W świecie urządzeń osadzonych wersja powinna być dostępna za pośrednictwem głównej ścieżki komunikacyjnej (USB, Wi-Fi, UART albo inna magistrala). Powinna ona być wypisywana automatycznie podczas rozruchu urządzenia. Jeśli nie jest to możliwe, postaraj się udostępniać ją na żądanie. Jeśli to również jest niemożliwe, powinna być kompilowana do własnego pliku obiektowego i umieszczana pod określonym adresem, żeby dało się poddać ją inspekcji.

Rozważ użycie *semantycznej kontroli wersji*, w której wersja ma postać *A.B.C*:

- *A* to główny numer wersji (1 bajt),
- *B* to poboczny numer wersji (1 bajt),
- *C* to wskaźnik kompilacji (2 bajty).

Jeśli wskaźnik kompilacji nie zwiększa się automatycznie, powinieneś często go zwiększać (cyferki są tanie). W zależności od metody wypisywania i estetyki systemu do kodu rejestrowania zdarzeń możesz dodać interfejs do poprawnego wyświetlania wersji:

```
void LogVersion(struct sFirmwareVersion *v)
```

Wykonywany kod to nie jedyna część systemu, która powinna mieć wersję. Każda część, która jest oddzielnie budowana lub aktualizowana, powinna mieć wersję stanowiącą część protokołu. Jeśli na przykład podczas produkcji programowana jest pamięć EEPROM, powinna mieć ona wersję sprawdzaną przez kod przed użyciem pamięci. Czasem nie masz wystarczająco dużo miejsca albo mocy przetwarzania, żeby Twój system był kompatybilny wstecz, ale jest niezwykle ważne, żeby wszystkie jego ruchome części były „kompatybilne teraz".

Projekty innych podsystemów nie zależą (i nie powinny zależeć) od sposobu, w jaki realizowane jest rejestrowanie. Jeśli możesz to powiedzieć o interfejsach swoich modułów („Inne podsystemy nie zależą od sposobu, w jaki realizowana jest funkcja XYZ; wywołują tylko dany interfejs"), to pomyślnie zaprojektowałeś interfejsy systemu.

Stan podsystemu rejestrowania

Kiedy będziesz pracował nad architekturą, niektóre obszary będą łatwiejsze do zdefiniowania niż inne, zwłaszcza jeśli wcześniej robiłeś coś podobnego. A kiedy będziesz definiował interfejsy, może się okazać, że przychodzą Ci do głowy pomysły, których implementacja byłaby łatwa (a nawet przyjemna), ale tylko pod warunkiem, że zacząłbyś *od razu*.

Wstrzymaj się chwilę, zanim ulegniesz pokusie zajęcia się implementacją; staraj się utrzymywać wszystko na tym samym poziomie. Jeśli dopieścisz jeden moduł przed zdefiniowaniem interfejsu drugiego, możesz odkryć, że niezbyt do siebie pasują.

Kiedy posuniesz się nieco dalej w pracy nad wszystkimi modułami w systemie, nadejdzie czas rozważenia stanu związanego z każdym modułem. Ogólnie rzecz biorąc, im mniej stanu przechowujesz w module, tym lepiej (żeby Twoje funkcje nie musiały robić tych samych rzeczy za każdym razem, kiedy je wywołujesz). Jednak całkowite wyeliminowanie stanu jest zwykle niewykonalne (a przynajmniej bardzo uciążliwe).

Wracając do naszego modułu rejestrowania: czy domyślasz się, jakiego będzie potrzebować wewnętrznego stanu? Funkcje `LogGlobalOn` i `LogGlobalOff` ustawiają (i zerują) jedną zmienną. Funkcja `LogSetOutputLevel` potrzebuje poziomu dla każdego podsystemu.

Masz kilka opcji zaimplementowania tych zmiennych. Jeśli chcesz wyeliminować lokalny stan, możesz umieścić je w strukturze (lub obiekcie), którą musiałaby mieć każda funkcja wywołująca moduł rejestrowania. Wymaga to jednak przekazywania obiektu rejestrowania do każdej funkcji, która mogłaby potrzebować rejestrowania, i przekazywania go do każdej funkcji mającej funkcję, która musi coś zarejestrować.

Może wydaje Ci się, że takie przekazywanie stanu jest zagmatwane. Jeśli chodzi o logowanie, trudno się z tym nie zgodzić. Czy zastanawiałeś się jednak, co jest w deskryptorze pliku, który otrzymujesz, kiedy Twój kod otwiera pliki? Otwarte pliki reprezentują mnóstwo informacji o stanie, w tym bieżące pozycje odczytu i zapisu, bufor oraz uchwyt do lokalizacji pliku na dysku.

Może przekazywanie wszystkich tych parametrów tam i z powrotem nie jest dobrym pomysłem. Jak każdy użytkownik podsystemu rejestrowania może uzyskać do nich dostęp? Jak wspomniano w ramce „Programowanie obiektowe w C", przy odrobinie dodatkowej pracy nawet

język C może tworzyć obiekty, które upraszczają ten problem przekazywania parametrów. Nawet w bardziej obiektowym języku mógłbyś mieć moduł, w którym funkcje są dostępne globalnie, a stan jest przechowywany w lokalnym obiekcie. Istnieje jednak inny sposób, aby zapewnić dostęp do obiektu logowania bez zupełnego otwierania dostępu do modułu.

Programowanie obiektowe w C

Dlaczego nie używać języka C++? Większość systemów wbudowanych ma całkiem dobre kompilatory C++. Istnieje jednak mnóstwo kodu napisanego w C, a czasem musisz dostosować się do tego, co już masz. A może po prostu lubisz język C dlatego, że jest szybki. Nie znaczy to, że powinieneś zrezygnować z zasad programowania obiektowego.

Jedną z najważniejszych koncepcji, które warto zachować, jest **ukrywanie danych**. W języku obiektowym Twój obiekt (klasa) może zawierać zmienne prywatne. Oznacza to, że ma on wewnętrzny stan, którego nie może zobaczyć nikt inny. C ma różne rodzaje zmiennych globalnych. Przez poprawne określenie zasięgu zmiennej możesz zasymulować ideę zmiennych prywatnych (a nawet klas zaprzyjaźnionych). Zacznijmy od odpowiednika zmiennej publicznej w języku C. Deklaruje się ją na zewnątrz funkcji, zwykle na początku źródłowego pliku C:

```
// Tę zmienną widzi każdy, kto deklaruje
// "extern tBoolean gLogOnPublic;"
tBoolean gLogOnPublic;
```

Właśnie z takich zmiennych globalnych powstaje kod spaghetti. Staraj się ich unikać. Zmienną prywatną deklaruje się za pomocą słowa kluczowego `static`. Robi się to na zewnątrz funkcji, zwykle na początku źródłowego pliku C:

```
// zmienne plikowe to zmienne globalne z odrobiną enkapsulacji
static tBoolean gLogOnPrivate;
```

Słowo kluczowe `static` znaczy różne rzeczy w różnych miejscach, co jest trochę irytujące. W przypadku funkcji i zmiennych poza funkcjami znaczy ono „ukryj mnie tak, żebym nie była widoczna w innych plikach” i ogranicza zasięg. W przypadku zmiennych wewnątrz funkcji słowo kluczowe `static` nakazuje zachowywać wartość zmiennej między kolejnymi wywołaniami. Staje się ona zatem zmienną globalną o zasięgu ograniczonym do funkcji, w której jest zadeklarowana.

Zbiór luźnych zmiennych jest nieco nieporęczny, więc można zdefiniować strukturę wypełnioną zmiennymi prywatnymi dla modułu:

```
// wszystkie zmienne globalne umieszczamy w strukturze:
struct {
  tBoolean logOn;
  enum eLogLevel outputLevel[NUM_LOG_SUBSYSTEMS];
} sLogStruct;
static struct sLogStruct gLogData;
```

Gdybyś chciał, żeby Twój kod C bardziej przypominał obiekt, struktura ta nie byłaby częścią modułu, ale tworzyłbyś ją (przydzielał wywołaniem `malloc`) podczas inicjalizacji (w przypadku systemu rejestrowania omawianego w tym rozdziale — w funkcji `LogInit`) i zwracał wywołującemu w następujący sposób:

```
struct sLogStruct* LogInit(){
  int i;
  struct sLogStruct *logData = malloc(sizeof(*logData));
  logData->logOn = FALSE;
  for (i=0; i < NUM_LOG_SUBSYSTEMS; i++) {
    logData-> outputLevel[i] = eNoLogging;
  }
  return logData;
}
```

Strukturę tę można przekazywać tam i z powrotem jak obiekt. Oczywiście, musiałbyś zapewnić sposób zwalniania pamięci zajętej przez obiekt; w tym celu wystarczyłoby dodać kolejną funkcję do interfejsu.

Wzorzec singleton

Aby upewnić się, że każda część systemu będzie miała dostęp do tego samego obiektu rejestrowania, można również posłużyć się innym wzorcem, tak zwanym *singletonem*.

Wzorca singletonu często używa się wtedy, kiedy jest ważne, aby klasa miała dokładnie jedną instancję. W językach obiektowych singleton jest odpowiedzialny za przechwytywanie żądań tworzenia nowych obiektów, aby zachować swój samotniczy stan. Dostęp do zasobu jest globalny (każdy może go używać), ale wszystkie dostępy przechodzą przez pojedynczą instancję. Nie ma publicznego konstruktora. W języku C++ wyglądałoby to tak:

```
class Singleton {
public:
  static Singleton* GetInstance() {
    if (instance_ == NULL) {
      instance_ = new Singleton;
    }
    return instance_;
  }
protected:
  Singleton(); // nikt nie może tworzyć obiektu tej klasy poza nią samą

private:
  static Singleton* instance_; // jedna jedyna instancja
};

// Definiujemy jeden jedyny wskaźnik do singletonu
Singleton* Singleton::instance_ = nullptr;
```

W przykładzie z rejestrowaniem zdarzeń singleton zapewnia całemu systemowi dostęp do podsystemu rejestrowania za pośrednictwem tylko jednej instancji. Kiedy masz pojedynczy zasób (taki jak port szeregowy), z którym muszą współdziałać różne części systemu, singleton bywa przydatny, ponieważ pozwala uniknąć konfliktów.

W językach obiektowych singletony umożliwiają też „leniwą” alokację i inicjalizację, więc moduły, które nigdy nie są używane, nie zużywają zasobów.

Globalne współdzielenie zmiennych prywatnych

Koncepcję singletonu można zastosować nawet w języku proceduralnym, takim jak C. Zalety ukrywania danych opisano we wcześniejszej ramce „Programowanie obiektowe w C". Ochrona zmiennych modułu przed modyfikacją (i użyciem) przez inne pliki sprawi, że Twoje rozwiązanie będzie bardziej niezawodne.

Czasem jednak potrzebujesz tylnych drzwi do informacji, albo dlatego, że musisz ponownie wykorzystać pamięć RAM w innym celu (rozdział 8.), albo dlatego, że musisz przetestować moduł od zewnątrz (rozdział 3.). W języku C++ mógłbyś użyć klasy zaprzyjaźnionej, aby uzyskać dostęp do zwykle ukrytych danych wewnętrznych.

W języku C, zamiast deklarować zmienne modułu jako naprawdę globalne przez usunięcie słowa kluczowego `static`, możesz nieco oszukać i zwrócić wskaźnik do zmiennych prywatnych:

```
static struct sLogStruct gLogData;
struct sLogStruct* LogInternalState() {
  return &gLogData;
}
```

Nie jest to dobry sposób na zachowanie enkapsulacji i ukrycie danych, więc używaj go oszczędnie. Rozważ zabezpieczenie się przed takim dostępem podczas normalnego działania kodu:

```
static struct sLogStruct gLogData;
struct sLogStruct* LogInternalState() {
#if PRODUCTION
  #error "Wewnętrzny stan rejestrowania jest chroniony!"
#else
  return &gLogData;
#endif /* PRODUCTION */
}
```

Kiedy będziesz projektował swój interfejs i zastanawiał się nad informacjami o stanie, które będą częścią Twoich modułów, pamiętaj, że będziesz również potrzebował metod do weryfikowania systemu.

Zabawa w piaskownicy

Omówiliśmy już prostokąty najniższego i średniego poziomu, ale musimy jeszcze przemyśleć prostokąty algorytmu. Jednym z celów dobrej architektury jest segregacja algorytmu. Doskonałym kandydatem jest tu standardowy wzorzec **modelu-widoku-kontrolera** (ang. *model-view-controller*, MVC). Celem wzorca jest odizolowanie sedna aplikacji od interfejsu użytkownika, żeby można było je rozwijać i testować niezależnie.

Zwykle wzorca MVC używa się po to, aby aplikacja mogła obsługiwać różne ekrany, interfejsy i platformy implementacyjne. Można go jednak również użyć po to, aby dało się rozwijać aplikację i algorytmy niezależnie od sprzętu. W tym wzorcu **widok** jest interfejsem użytkownika, który pełni funkcję zarówno wejścia, jak i wyjścia. W naszym urządzeniu użytkownik nie musi być człowiekiem; mogą to być czujniki sprzętowe (wejście) i ekran (wyjście). W rzeczywistości,

jeśli masz system, który nie ma ekranu, ale przesyła dane przez sieć, widok może nie mieć aspektu wizualnego, ale nadal jest częścią systemu jako forma wejścia-wyjścia. **Model** to dane i logika specyficzne dla dziedziny problemu. Jest to część, która przyjmuje surowe dane z wejścia i przekształca je w coś użytecznego, często z użyciem algorytmu, który sprawia, że Twój produkt jest wyjątkowy. **Kontroler** jest klejem, który spaja model i widok; odpowiada za dostarczanie danych wejściowych do modelu w celu przetworzenia oraz przekazywanie danych wyjściowych z modelu na ekran albo do zewnętrznego łącza komunikacyjnego. Są to standardowe sposoby interakcji tych trzech elementów, ale na razie wystarczy pamiętać o podziale funkcji.

Istnieje inny sposób użycia wzorca MVC, który bardzo przydaje się podczas tworzenia i weryfikowania algorytmów dla systemów wbudowanych: użycie *urządzenia wirtualnego*, czyli **piaskownicy**. Im bardziej złożony algorytm, tym bardziej jest to konieczne.

Jeśli możesz wziąć wszystkie wejścia do prostokąta algorytmu i przepuścić je przez jeden interfejs, możesz przełączać się tam i z powrotem między plikiem na komputerze a rzeczywistym systemem.

Podobnie, jeśli możesz przekierować wyniki algorytmu do pliku, możesz testować algorytm na komputerze (gdzie środowisko diagnostyczne może być o wiele lepsze niż w systemie wbudowanym) i wielokrotnie przetwarzać te same dane, aż zidentyfikujesz i poprawisz wszystkie błędy. Jest to doskonały sposób na przeprowadzanie testów regresyjnych w celu upewnienia się, że niewielkie zmiany algorytmu nie mają nieprzewidzianych skutków ubocznych.

Różne aspekty modelu-widoku-kontrolera

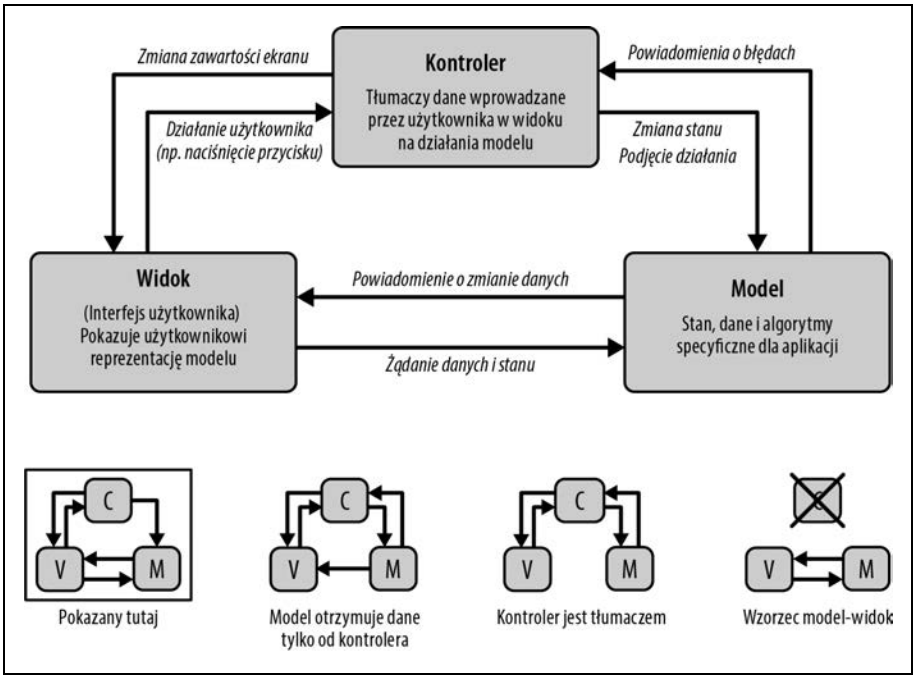
Jeśli chodzi o wzorzec MVC, dobra wiadomość jest taka, że niemal wszyscy zgadzają się, że ma on trzy części. Zła wiadomość jest taka, że jest to prawdopodobnie jedyna rzecz, z którą wszyscy się zgadzają.

Model przechowuje dane, stan i logikę aplikacji. Jeśli budujesz stację pogodową, model zawiera kod do monitorowania temperatury i modele predykcyjne. W przypadku odtwarzacza MP3 model składa się z bazy utworów oraz kodeka potrzebnego do ich odtwarzania.

Widok, tradycyjnie rozważany w kontekście urządzenia z ekranem, reprezentuje funkcje obsługi ekranu. Widok to sposób, w jaki użytkownik widzi model. Widok może być obrazkiem słońca albo szczegółowymi odczytami statystyk z usługi meteorologicznej. Są to dwa widoki tych samych informacji.

Kontroler to dość nieokreślona chmura, która znajduje się pomiędzy modelem i widokiem i pomaga im ze sobą współdziałać. Celem kontrolera jest uniezależnienie widoku od modelu, żeby każdy z nich można było wykorzystać ponownie. Na przykład w przypadku odtwarzacza MP3 firma może chcieć, żeby interfejs pozostał taki sam nawet po wprowadzeniu nowej wersji sprzętu. A może jest to ten sam sprzęt, ale dział marketingu chce, żeby system wyglądał na bardziej przyjazny dla dzieci. Jak osiągnąć te cele przy zmianie jak najmniejszej ilości kodu? Kontroler umożliwia rozdzielenie modelu i widoku, zapewniając takie usługi jak tłumaczenia naciśnięcia przycisku na zdarzenie dla modelu.

Na rysunku 2.9 pokazano podstawową interpretację wzorca MVC i kilka typowych wariantów. Istnieje wiele sposobów łączenia poszczególnych elementów, przy czym niektóre są pozornie sprzeczne. Termin „MVC" jest dość mglisty i trzeba znać kontekst, żeby wiedzieć, co oznacza on w danym przypadku.

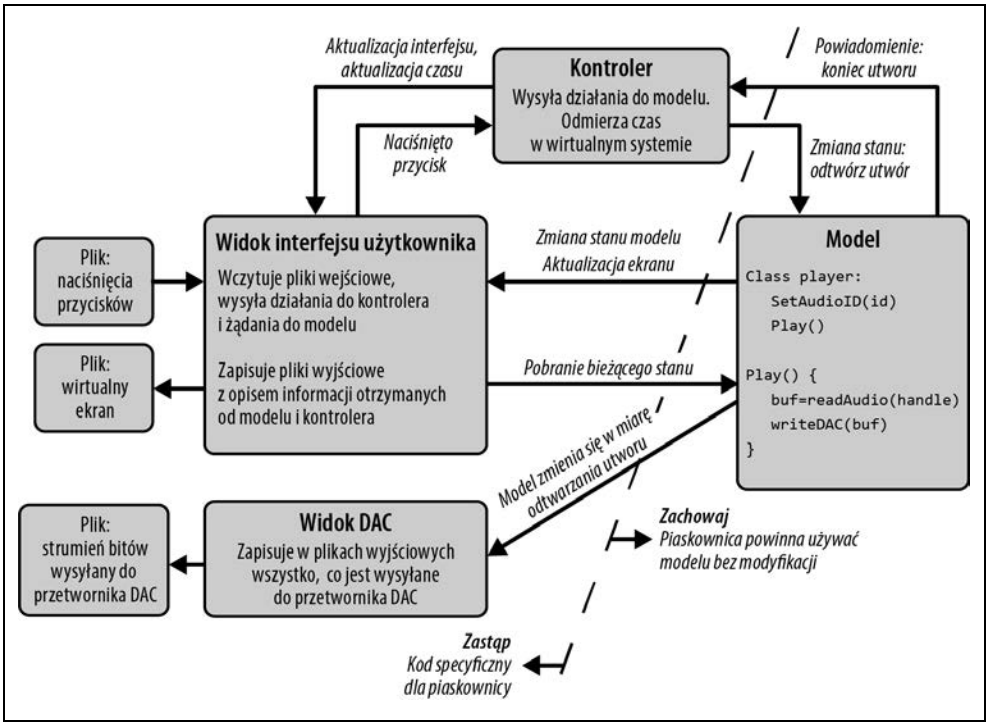


Rysunek 2.9. Przegląd wzorca model-widok-kontroler

W środowisku piaskownicy Twoje pliki i sposób wczytywania danych należą do widoku wzorca MVC. Algorytm, który testujesz, jest modelem i nie powinien się zmieniać. Kontroler jest częścią, która się zmienia, kiedy umieszczasz algorytm w komputerze. Rozważ odtwarzacz MP3 (rysunek 2.10) i diagram MVC w przypadku użycia piaskownicy. Zauważ, że zarówno widok, jak i kontroler mają dość rygorystycznie zdefiniowany interfejs API, ponieważ zastąpisz je, kiedy przejdziesz od urządzenia wirtualnego do rzeczywistego.

Twój wejściowy plik widoku może przypominać scenariusz filmowy i instruować piaskownicę tak, aby zachowywała się jak potencjalny użytkownik. Pierwsze pole określa czas, w którym w Twoim systemie aktywowana jest metoda określona w drugim polu:

```
Czas, Działanie, Zmienne argumenty // komentarz
00:00.00, PowerOnClean             // przeprowadź inicjalizację
00:01.00, PlayPressed              // brak działania, nie wczytano utworu
00:02.00, Search, "Still Alive"    // oczekujemy listy pasujących utworów
00:02.50, PlayPressed              // oczekujemy odtworzenia utworu
```



Rysunek 2.10. Model-widok-kontroler w piaskownicy

Plik wyjściowy może wyglądać tak, jak tylko chcesz. Możesz na przykład mieć wiele plików wyjściowych, jeden opisujący stan i zmiany wysyłane do interfejsu użytkownika z kontrolera i modelu, a drugi zawierający to, co model wysyłałby do przetwornika cyfrowo-analogowego (ang. *digital-to-analog converter*, DAC). Pierwszy plik wyjściowy mógłby wyglądać tak:

```
Czas, Podsystem, Komunikat dziennika
00:00.01, System, Inicjalizacja: piaskownica wersja 1.3.45
00:01.02, Kontroler, pomniejszy błąd: brak wczytanego utworu
00:02.02, Kontroler, 4 utwory dla wyszukiwania "Still Alive", wybrano ID 0x1234
00:02.51, Kontroler, wczytywanie klasy player w celu odtworzenia ID 0x1234
```

Ponownie, sam definiujesz format tak, aby spełniał Twoje potrzeby (które zapewne się zmienią). Jeśli na przykład Twój odtwarzacz ma usterkę i odtwarza niewłaściwy utwór, piaskownica mogłaby wypisywać w każdym wierszu identyfikator utworu.

Model-widok-kontroler to wzorzec bardzo wysokiego poziomu, mówiąc ściślej — poziomu systemowego. Kiedy zaczniesz się zastanawiać nad dzieleniem systemu w taki sposób, możesz odkryć, że ma on własności fraktalne. A gdybyś na przykład chciał tylko przetestować kod, który wczytuje plik z pamięci i wysyła go do przetwornika DAC? Nazwa pliku i wynikowy strumień bitowy byłyby widokiem (który składa się z wejść i wyjść, nawet jeśli nie korzysta z niego żaden człowiek), logika i dane potrzebne do przekształcenia pliku byłyby modelem, a kod do obsługi plików byłby kontrolerem, ponieważ ta część musiałaby się zmieniać w zależności od platformy.

Mając na uwadze te idee separacji i wykonywania kodu w piaskownicy, przyjrzyj się rysunkom architektonicznym i sprawdź, ile wejść ma algorytm. Jeśli ma więcej niż jedno, czy ma sens utworzenie specjalnego obiektu interfejsu? Czasem lepiej jest mieć wiele plików, które będą reprezentować wiele zachodzących procesów. Przyglądając się dokładniej diagramom, czy ma sens zastosowanie piaskownicy na poziomie niższym niż algorytmy odpowiadające za funkcje produktu? Poszerza to Twój model i, co ważniejsze, pozwala przetestować więcej kodu w kontrolowanym środowisku.

Tworzenie piaskownicy może być kosztowne, więc lepiej nie zabieraj się od razu za jej implementowanie. W szczególności piaskownica może używać zmiennych innej wielkości, a jej kompilator może działać nieco inaczej, więc wstrzymaj się z jej budowaniem, chyba że Twoje algorytmy są bardzo skomplikowane albo wiesz, że będziesz musiał długo czekać na fizyczny sprzęt. Istnieje kompromis między czasem, który trzeba poświęcić na budowanie piaskownicy, a oferowanymi przez nią niezrównanymi możliwościami debugowania i testowania. Jeśli jednak na podstawie projektu ustalisz, jak zbudować piaskownicę, i będziesz o tym pamiętać podczas rozwijania swojej architektury, będziesz mieć wyjście awaryjne, jeśli coś pójdzie nie tak.

Z powrotem do tablicy

Zaleciłam Ci tworzenie szkiców systemu. Pamiętaj, szkice są tanie! Możesz wypróbować różne pomysły i opcje, zanim podejmiesz ostateczną decyzję. Dużo łatwiej zmienić szkice niż kod źródłowy całego systemu.

Kiedy patrzysz na swoje szkice, pamiętając o enkapsulacji, ukrywaniu danych, interfejsach i wzorcach projektowych, co widzisz? Rysunki, które wybrałam, mają dać Ci różne punkty widzenia na system. Zadają pytania dotyczące architektury, żebyś mógł przemyśleć potencjalne problemy na wcześniejszych etapach procesu, podczas projektowania, a nie programowania.

Żaden projekt nie pozostaje statyczny. Celem projektu jest zrozumienie rzeczy z perspektywy systemu. Nie oznacza to, że musisz aktualizować swoje diagramy po każdej zmianie. Niektóre diagramy staną się częścią dokumentacji i trzeba będzie je aktualizować, ale szkice są dla Ciebie — mają pomagać Ci w zrozumieniu systemu oraz w komunikacji z zespołem.

Diagram kontekstowy to mapa „jesteś tutaj”, która wskazuje, co powinieneś robić. Zapewne będzie użyteczna podczas rozmów o wymaganiach wobec systemu.

Diagram blokowy pokazuje elementy sprzętowe i programowe. Widoczne są na nim dostępne zasoby wraz z krótkim opisem ich zamierzonego użycia. Panoramiczny widok pomaga w identyfikowaniu zadań, które trzeba będzie zrealizować. Warto zacząć od niespecyficznych prostokątów; reprezentowane przez nie układy scalone zapewne się zmienią, więc jak możesz uodpornić swój kod na takie zmiany? Diagram blokowy to najczęściej używany rysunek, który zapewne okaże się bardzo przydatny podczas rozmów ze współpracownikami.

Organigram pokazuje przepływ sterowania oraz potencjalne konflikty w miejscach, w których wiele podsystemów próbuje uzyskać dostęp do tych samych zasobów. Może pomóc w enkapsulacji modułów, które nie muszą się ze sobą komunikować (albo muszą, ale nie powinny robić tego bezpośrednio). Nie jest to łatwe, ale mnie pomaga w tym metafora kierownika: kto

wydaje polecenia temu komponentowi? Diagram ten pomaga zrozumieć szczegóły systemu i diagnozować dziwne błędy.

Diagram warstwowy pokazuje, jak można zgrupować moduły, aby utworzyć interfejsy i warstwy abstrakcji. Pomaga określić, które informacje mogą być ukryte. Pozwala też zidentyfikować moduły, które są za duże i powinny być podzielone na mniejsze części. Widok ten może pomóc w ograniczeniu złożoności systemu i pisaniu kodu nadającego się do ponownego wykorzystania (albo zrozumieniu warstw abstrakcji sprzętowej dostarczonych przez producenta procesora).

Każdy z tych diagramów był prezentowany na poziomie całego systemu. Systemy składają się jednak z podsystemów, które składają się z podsystemów. Jeśli diagram wysokiego poziomu jest zbyt szczegółowy, możesz stracić z oczu widok całego systemu. Czasem duży prostokąt trzeba przenieść na nową stronę i tam uzupełnić go o szczegóły (i gruntowniej przemyśleć w miarę projektowania i coraz lepszego rozumienia tego podsystemu).

Jeśli diagramy wydają Ci się użyteczne, ale nie jesteś przekonany do tych, które opisałam w Tym rozdziale, poszukaj informacji o modelu C4 do wizualizacji architektury oprogramowania albo o **modelu widoku architektonicznego 4 + 1** — oba są doskonałymi alternatywami. Jeśli chodzi o program do rysowania, kiedy jestem gotowa do sformalizowania moich szkiców, używam usługi mermaid.live, ponieważ pozwala ona definiować diagramy w postaci tekstu, który można umieścić w większości dokumentów Markdown.

Dalsza lektura

W tym rozdziale wspomniałam o kilku spośród licznych wzorców projektowych. Będę o nich mówić również w kolejnych rozdziałach, ale ta książka jest poświęcona systemom wbudowanym, a nie wzorcom projektowym. Aby dowiedzieć się więcej o standardowych wzorcach oprogramowania, sięgnij po jedną z poniższych pozycji:

- Erich Gamma i in., *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku* (Helion). Jest to pierwsza, niezwykle wpływowa praca poświęcona wzorcom projektowym. Językiem odniesienia jest w niej C++.
- Eric Freeman i in., *Wzorce projektowe. Rusz głową! Tworzenie rozszerzalnego i łatwego w utrzymaniu oprogramowania obiektowego* (Helion). Ta książka używa Javy jako przykładowego języka. Zawiera doskonałe przykłady i jest utrzymana w zajmującym stylu.
- Wyszukaj hasło „Wzorce projektowe” w Wikipedii.

Podałam przegląd architektury i metod jej szkicowania, ale jeśli Twój system jest bardziej złożony albo Twoje diagramy mają zostać wykorzystane w procesie certyfikacji bezpieczeństwa, oto inne dobre zasoby:

- Len Bass i in., *Documenting Software Architectures* (Addison-Wesley). To świetna książka, która omawia różne style architektoniczne i odpowiednie modele wizualne.
- Hassan Gomaa, *Real-Time Software Design for Embedded System* (Cambridge University Press). Omawia różne modele i diagramy, za pomocą których możesz rozrysować szkic swojego systemu.

Choć tworzeniu modułów przyjrzymy się nieco później, jeśli chcesz przeskoczyć do przodu albo jesteś zainteresowany rejestrowaniem zdarzeń i nie wystarcza Ci krótki kod, który zamieściłam w tym rozdziale, rzuć okiem na stronę Arduino Logging Library w witrynie Embedded Artistry (*https://embeddedartistry.com/arduino-logger*). Autor opisuje na niej proces tworzenia biblioteki i udostępnia kod, którego możesz użyć we własnym projekcie. Kiedy już tam będziesz, kliknij łącze *Welcome* (*https://oreil.ly/siMLU*), ponieważ witryna Embedded Artistry zawiera mnóstwo interesujących informacji i przykładów kodu.

Pytanie rekrutacyjne: stwórz architekturę

Opisz architekturę [przedmiotu w pokoju].

Rozglądanie się po miejscu rozmowy kwalifikacyjnej w poszukiwaniu odpowiedniego przedmiotu jest nieco problematyczne, ponieważ w takich miejscach zwykle jest niewiele ciekawych rzeczy. Często wybieram telefon konferencyjny, który bywa najbardziej skomplikowanym systemem w pokoju. Innym dobrym przykładem jest projektor.

Kiedy zadaję to pytanie, chcę sprawdzić, czy kandydat potrafi podzielić problem na części. Interesuje mnie, jak przeprowadzi proces mentalnej dekonstrukcji obiektu. Ogólnie rzecz biorąc, najbezpieczniej jest zacząć od wejść i wyjść. W przypadku telefonu konferencyjnego wyjściami są głośnik i ekran, wejściami — przyciski i mikrofon. Chciałabym zobaczyć je jako prostokąty na kartce papieru. Kandydat nie powinien się obawiać podnieść przedmiot i obejrzeć jego połączenia. To również są wejścia i wyjścia. Kiedy kandydat rozrysuje fizyczny sprzęt, może zacząć dodawać połączenia, zadając (sobie) pytania, jak działa każdy komponent: jak działa przycisk zasilania i jaki może być interfejs między nim a oprogramowaniem? Jak działa mikrofon i co to może oznaczać dla innych części systemu (np. czy w telefonie jest przetwornik analogowo-cyfrowy)?

Kandydat dostaje punkty za wspomnienie o dobrych zasadach projektowania oprogramowania. Dobrze, żeby prostokąty na najniższym poziomie nazywał sterownikami, a prostokąty na kolejnym poziomie — obiektami. Dobrze też, jeśli wspomina o częściach systemu, które można wykorzystać ponownie w przyszłym modelu telefonu, i o tym, jak zachować ich enkapsulację.

Oczekuję, że zada pytania o konkretne funkcje albo cele projektowe (takie jak koszt). Ma jednak sporo swobody, jeśli chodzi o szczegóły. Chce rozmawiać o sieci? Niech udaje, że jest to telefon VoIP. Chce zupełnie pominąć ten temat? Niech skupi się na tym, jak telefon może przechowywać numery w miniaturowej bazie danych albo na połączonej liście. Jestem zadowolona, kiedy mówi o rzeczach, które go interesują, zwłaszcza o obszarach, o które nie pytałam w innych częściach rozmowy kwalifikacyjnej.

Podobnie, choć interesuje mnie ogólna panorama całego systemu, nie mam nic przeciwko, żeby kandydat zagłębił się w pewien obszar. Pytanie to daje mu dużą swobodę w tłumaczeniu, jak jego konkretne doświadczenie pomogłoby mu w projektowaniu telefonu. Kiedy je zadaję, nie przeszkadza mi, jeśli kandydat przyzna się do częściowej niewiedzy i opowie o czymś, na czym zna się lepiej.

Gdy pytam o architekturę, nie oczekuję, że kandydat poda mi idealne szczegóły techniczne. Najważniejsze, żeby coś narysował, nawet jeśli jest to ledwo czytelne. Chcę, aby wykazał się entuzjazmem w rozwiązywaniu problemów i umiejętnością efektywnego przekazywania swoich pomysłów.

Dzięki książce zorientujesz się w zawiłościach procesów i wzorców budowy oprogramowania wbudowanego!

Miro Samek, znawca systemów wbudowanych, autor książek i nauczyciel

Systemy wbudowane napędzają działanie urządzeń medycznych, samochodów, samolotów, sprzętów AGD, a nawet zabawek dla dzieci. Zazwyczaj pracują w środowiskach o ściśle określonych parametrach sprzętowych i często nie korzystają ze wsparcia systemów operacyjnych. Dlatego ich tworzenie wymaga dużej precyzji, a także odmiennego podejścia do projektowania i implementacji oprogramowania.

Lektura tej książki pozwoli Ci przyswoić kluczowe koncepcje i opanować dobre praktyki, które warto stosować podczas tworzenia kodu. Poznasz zarówno klasyczne wzorce projektowe, jak i te opracowane specjalnie z myślą o systemach wbudowanych. Znajdziesz tu rozdziały poświęcone nowoczesnym technologiom, takim jak systemy współpracujące z internetem rzeczy i czujniki sieciowe, a także omówienie zagadnień związanych z silnikami. Dokładnie zbadasz tematykę debugowania, strategii zarządzania danymi — i wiele więcej! Dowiesz się, jak budować architekturę urządzenia z uwzględnieniem procesora, a nie systemu operacyjnego. Zapoznasz się również z technikami rozwiązywania problemów sprzętowych, modyfikowania projektów i definiowania wymagań produkcyjnych.

Najciekawsze zagadnienia:

- optymalizacja systemu pod kątem kosztów i wydajności
- zapewnianie niezawodności w środowisku o ograniczonych zasobach
- czujniki, wyświetlacze, silniki i inne urządzenia wejścia-wyjścia
- redukcja zużycia pamięci RAM, przestrzeni kodu, cykli procesora i energii
- projektowanie systemów wbudowanych współdziałających z internetem rzeczy i czujnikami sieciowymi

Elecia White pracowała nad monitorami intensywnej terapii, systemami dla lotnictwa i samochodów wyścigowych, zabawkami edukacyjnymi i wieloma innymi urządzeniami. Obecnie pełni funkcję głównego inżyniera do spraw systemów wbudowanych. Współprowadzi Embedded.fm — podcast poświęcony systemom wbudowanym i kreatywnym technologiom.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1829-0
9 788328 918290
Cena: 89,00 zł