# Politechnika Koszalińska Kierowca

# Cel

Celem projektu jest stworzenie serwisu “PKKierowca” służącego do rejestrowania aktualnej pozycji i prędkości pojazdów we flocie firmy.

# Rynek

W odróżnieniu od innych serwisów serwis PKKierowca będzie obsługiwał wyłącznie samochody pracowników PK w celu prowadzenia różnych badań na temat stylu prowadzenia samochodu przez ludzi .

# Użytkownicy

Użytkownikiem systemu będą pracownicy korzystający z samochodów firmowych oraz ich przełożeni.

1. Opracowanie ogólnego planu pracy zespołu:
   1. Organizacja zespołu

Scrum:

1. **Scrum Master** - osoba czuwająca nad tym, żeby były przestrzegane wszystkie zasady Scruma. Do jej obowiązków należy również pomoc w rozwiązywaniu problemów organizacyjnych.
2. **Product Owner** — osoba reprezentująca klienta i dbająca o całokształt projektu. Bardzo często jest ona członkiem zespołu i jest pośrednikiem pomiędzy zespołem deweloperskim a klientem.
3. **Zespół deweloperski** — zespół odpowiedzialny za implementacje aplikacji. Przeważnie nie wyróżniamy tutaj żadnego lidera, a więc struktura zespołu jest płaska:

* programiści
* tester – współpracuje z klientem przy pisaniu scenariuszy i uruchamia testy sprawdzające realizacje scenariuszy
* śledzący (tracker) – sprawdza postęp prac oraz weryfikuje wszelkie oszacowania
* Dokumentacja (przy 4-osobowym zespole min 1 osoba i druga do pomocy)

Programowanie ekstremalne:

1. programiści
2. tester – współpracuje z klientem przy pisaniu scenariuszy i uruchamia testy sprawdzające realizacje scenariuszy
3. konsultant (okazjonalnie - wykładowca)
4. Dokumentacja (przy 4-osobowym zespole min 1 osoba i druga do pomocy)
   1. Metodologia pracy

**Scrum**

Proces realizacji projektu z wykorzystaniem metodyki Scrum można w uproszczeniu przedstawić następująco:

* **programiści**
* **tester** – współpracuje z klientem przy pisaniu scenariuszy i uruchamia testy sprawdzające realizacje scenariuszy
* **śledzący (tracker)** – sprawdza postęp prac oraz weryfikuje wszelkie oszacowania
* **trener (coach)** – nadzoruje proces tworzenia oprogramowania przez zespół, pomaga, kontroluje, itp.
* **konsultant (okazjonalnie)**

W scrumie mamy podział na 3 główne role:

**Scrum Master** — osoba czuwająca nad tym, żeby były przestrzegane wszystkie zasady Scruma. Do jej obowiązków należy również pomoc w rozwiązywaniu problemów organizacyjnych.

**Product Owner** — osoba reprezentująca klienta i dbająca o całokształt projektu. Bardzo często jest ona członkiem zespołu i jest pośrednikiem pomiędzy zespołem deweloperskim a klientem.

**Zespół deweloperski** — zespół odpowiedzialny za implementacje aplikacji. Przeważnie nie wyróżniamy tutaj żadnego lidera, a więc struktura zespołu jest płaska.

Plusy:

* Wykładowca się jej spodziewa, ponieważ na takim pracują w firmie i będzie miało to niewątpliwie plus jeśli chodzi o ocenianie oraz możliwe zatrudnienie (tak insynuuje każda jego wypowiedź na temat projektu)
* Scrum zawiera w sobie cechy metodyk zwinnych. Jest to najbardziej znane podejście do tworzenia oprogramowania i na pewno prędzej czy później się z nim spotkacie.
* Możliwość bieżącego nanoszenia zmian w projekcie w czasie jego realizacji - zakres projektu jest otwarty przez cały okres jego realizacji. Klient może więc nanosić zmiany zarówno w obrębie istniejących, jak i przyszłych funkcjonalności.
* Możliwość bieżącej regulacji zakresu projektu i zamknięcia go w dowolnym momencie - jeśli na skutek np. okrojonego budżetu projektu chcielibyśmy zrezygnować z realizacji części funkcjonalności, jest to jak najbardziej możliwe. To klient decyduje który sprint kończy dany projekt.
* Bieżący monitoring i ocena powstającego produktu - po każdym sprincie klient otrzymuje nową wersję produktu, którą może przetestować i nanieść swoje korekty, możliwe do zrealizowania w kolejnym przyroście.
* Formalności i biurokracja zmniejszone do minimum - dokumentacja projektowa nie jest niezbędna (wystarczą wymagania klienta i bieżący kontakt z nim), nie musimy więc ponosić kosztów jej wytworzenia.
* Ponoszenie wydatków wyłącznie za wykonaną pracę - faktyczny czas realizacji zagadnień przez firmę wdrożeniową może się nieco różnić od szacowanego. Klient ponosi koszty rzeczywistego czasu realizacji - co działa oczywiście zarówno w przypadku szybszego, jak i bardziej czasochłonnego wykonania zadania.
* Szybsza i skuteczniejsza realizacja projektu - dzięki bieżącym korektom kierunku rozwoju produktu, powstaje on szybciej i jest lepiej dopasowany do potrzeb klienta.

Minusy:

* **Brak wiedzy o dokładnym koszcie realizacji projektu** - bazujemy na zarysie funkcjonalnym projektu i szacunkowym koszcie realizacji. Rzeczywisty koszt zależy od końcowych założeń i faktycznym czasie ich realizacji przez firmę wdrożeniową.
* **Ponoszenie wszelkich kosztów związanych z projektem** - jeśli firmie wdrożeniowej z jakichś przyczyn nie uda się zrealizować zagadnienia w szacowanym czasie, rzeczywisty koszt stworzenia funkcjonalności rośnie - i my, jako klient musimy go pokryć. Oczywiście, tak jak sygnalizowałem wyżej, działa to w obie strony.
* **Ryzyko rozrastania się projektu** - mając możliwość zmian i dokładania kolejnych funkcjonalności w trakcie realizacji projektu, bez należytej kontroli, możemy doprowadzić do jego niechcianego i kosztownego rozrastania się.
* **Ryzyko przekroczenia zakładanego budżetu** - punkt będący konsekwencją powyższych. W przypadku niedostatecznej kontroli nad projektem, wymaganiami i jego budżetem, zakładany budżet szybko się wyczerpie, a produkt nie będzie realizował kluczowych funkcjonalności.

**Programowanie ekstremalne**

* **Brak planowania z góry** — nie ustalamy na sztywno żadnej konkretnej architektury systemu. Oczywiście przed rozpoczęciem planu definiujemy jakieś wzorce, jednak przyjmujemy do wiadomości, że architektura ulegnie zmianie w ramach rozwijania projektu.
* **Ciągła modyfikacja architektury** — ten punkt jest związany z poprzednim. Chodzi o to, aby, nie bać się ruszać architektury, jeśli ma to nam pomóc w dalszym rozwoju projektu. Trzeba pisać też kod w taki sposób, aby takie modyfikacje nie ciągnęły za sobą zmian w większości klas projektu.
* **TDD (programowanie napędzane testami)** — testy jednostkowe (pisałem o nich w odcinku o testowaniu) powinny być pisane jeszcze zanim powstanie kod, który mają testować. Dopiero później piszemy kod, który musi zaliczyć wszystkie te testy. Takie podejście sprawia, że pokrycie kodu testami jest bardzo duże, a dodatkowo wymusza na nas pisania kodu, do którego da się łatwo napisać testy.
* **Programowanie parami** — programiści powinni siedzieć parami. Jeden zajmuje się pisaniem kodu, drugi z nich analizuje cały czas kod oraz zgłasza jakieś poprawki. Co kilkadziesiąt następuje zamiana rolami. Ten punkt jest dość kontrowersyjny. Z jednej strony takie podejście zwiększa czytelność kodu i minimalizuje ryzyko pomyłki. Z drugiej strony potrzebujemy znacznie więcej programistów.
* **Iteracyjność** — pracujemy w cyklach o określonej przez nas długości. Na początku zakładamy, co uda nam się zrobić w danym cyklu. Po upłynięciu czasu wysyłamy klientowi aktualną wersję aplikacji oraz planujemy, co zrobimy w kolejnym cyklu.
* **Kontakt z klientem** — nie da się ustalić wszystkich szczegółów przed rozpoczęciem projektu. Dlatego ważne jest, aby wszelkie wątpliwości rozwiewać poprzez bezpośredni kontakt z klientem.

Od siebie dodam, iż jest to nasz najbardziej ulubiony model pracy. Z ostatnich naszych projektów to wynika. Ciężko nam coś zaplanować z góry, wolimy na bieżąco rozwijać projekt oraz na przestrzeni produkcji wpadamy na nowe pomysły i ulepszenia.

Jednakże nie sądzę żeby taki model przypadł do gustu „GLOBALOWI”, tam stawiają na planowanie i sumienność w wyznaczonych na początku działaniach.

## **Metodyki zwinne**

* Być może spotkaliście się ze słowem Agile, które z angielskiego oznacza zwinny. Jest to też nazwa jednej z popularniejszej metodyki prowadzenia projektu. O jej popularności może świadczyć fakt, że teraz każda firma reklamuje się pracowaniem według zasad agile.
* Cała idea polega na samo organizujących się zespołach. Oznacza to, że przykładowo zespół mobilny sam dobiera sobie odpowiednie narzędzia i wybiera architekturę odpowiednią dla danego projektu. Podobnie jak w programowaniu ekstremalnym, miarą postępów projektu jest działająca wersja aplikacji. Osiągamy to poprzez regularne wydawanie kolejnych wersji programu do klienta.
* Bardzo ważną kwestią jest także ciągła poprawa. Zespoły powinny aranżować spotkania, na których poddadzą analizie ostatnio wykonane postępy. Na podstawie wyciągniętych wniosków, wszyscy powinni wspólnie pomyśleć, co można by poprawić na przyszłość. W metodyce Agile bardzo ważna jest odpowiednia komunikacja. Najlepiej, jeśli odbywa się ona twarzą w twarz, chociaż zdarzają się też zespoły, w których każdy z pracowników pracuje z własnego domu

1. Stworzenie ogólnej Specyfikacji wymaga Systemu:
   1. Analiza wymagań
      1. Funkcjonalne
         1. Funkcje systemu

* Zakładanie kont kierowców.
* Definiowanie pojazdów.
* Przypisywanie kierowców do pojazdów (historia przypisań).
* Raportowanie aktualnej pozycji i prędkości.
* Detekcja przekroczeń prędkości.
* Interfejs Web API dla przeglądarek internetowych i aplikacji mobilnych.
  + - 1. Oferowane usługi
* Przeglądanie danych i wyszukiwanie aktualnej i historycznej pozycji pojazdów.
* Przebieg dzienny auta.
* Najczęściej użyte miejsca postoju.
* Najdłuższa oraz najkrótsza trasa z/do pracy po km oraz czasie.
* Zużycie paliwa miesięczne według dłuższa i krótsza trasa (obliczane z ilości tankowań na miesiąc).
* Podpowiedzi przy wyborze trasy na podstawie danych z najdłuższa i najkrótsza trasa z zeszłego tygodnia tego dnia.
  + - 1. Reakcje na dane wejściowe
* Obliczenia zużycia paliwa,
* Obliczenia tras,
* Obliczenia zużycia paliwa,
* Wnioskowanie którą trasę wybrać,
* Podpowiedzi najekonomiczniejszego kierowny.
  + 1. Niefunkcjonalne

1. Pobieranie danych z gps,
2. Przetwarzanie danych położenie, spalanie, postoje itp.
3. Obliczanie,
4. Zaznaczanie gps miejsc postoju,
5. Wyznaczanie najlepszej trasy,
6. Zbieranie danych o czasie,
7. Pobieranie/Zapis danych z bazy NoSql,
   * 1. Diagram przypadków użycia

* + 1. Model danych

NoSql:

Nierelacyjne bazy danych doskonale wpisują się w trend Big Data, ponieważ – w przeciwieństwie do klasycznych silników – pozwalają na szybką analizę danych niestrukturyzowanych i badanie korelacji pomiędzy nimi. W tradycyjnej bazie schemat i relacje są w narzucone z góry a za pomocą odpowiednich zapytań SQL możemy uzyskać strukturalne odpowiedzi mieszczące się we wcześniej opisanych ramach. Przykładowo w banku możemy znaleźć dane o kliencie np. w systemie kredytowym i CRM jeśli tabelę z obu powiążemy np. za pomocą pola PESEL. Dzięki relacji między tabelami będzie można otrzymać informacje, ile osób z danej ulicy – czy danego miasta – posiada kredyt większy niż 100 tys. zł. Będzie to możliwe, ponieważ tabelę zawierającą nazwę ulicy w systemie CRM z tabelą z systemu kredytowego łączy właśnie – nawet jeśli nie bezpośrednio – informacja o numerze ewidencyjnym PESEL.

5 typów baz danych nierelacyjnych:

* **Column:** Accumulo, Cassandra, Druid, HBase, Vertica, SAP HANA
* **Document:** Apache CouchDB, ArangoDB, Clusterpoint, Couchbase, DocumentDB, HyperDex, IBM Domino, MarkLogic, MongoDB, OrientDB, Qizx, RethinkDB
* **Key-value:** Aerospike, ArangoDB, Couchbase, Dynamo, FairCom c-treeACE, FoundationDB, HyperDex, InfinityDB, MemcacheDB, MUMPS, Oracle NoSQL Database, OrientDB, Redis, Riak, Berkeley DB
* **Graph:** AllegroGraph, ArangoDB, InfiniteGraph, Apache Giraph, MarkLogic, Neo4J, OrientDB, Virtuoso, Stardog
* **Multi-model:** Alchemy Database, ArangoDB, CortexDB, Couchbase, FoundationDB, InfinityDB, MarkLogic, OrientDB

# ińska Kierowca

# Cel

Celem projektu jest stworzenie serwisu “PKKierowca” służącego do rejestrowania aktualnej pozycji i prędkości pojazdów we flocie firmy.

# Rynek

W odróżnieniu od innych serwisów serwis PKKierowca będzie obsługiwał wyłącznie samochody pracowników PK w celu prowadzenia różnych badań na temat stylu prowadzenia samochodu przez ludzi .

# Użytkownicy

Użytkownikiem systemu będą pracownicy korzystający z samochodów firmowych oraz ich przełożeni.

1. Opracowanie ogólnego planu pracy zespołu:
   1. Organizacja zespołu

Scrum:

1. **Scrum Master** - osoba czuwająca nad tym, żeby były przestrzegane wszystkie zasady Scruma. Do jej obowiązków należy również pomoc w rozwiązywaniu problemów organizacyjnych.
2. **Product Owner** — osoba reprezentująca klienta i dbająca o całokształt projektu. Bardzo często jest ona członkiem zespołu i jest pośrednikiem pomiędzy zespołem deweloperskim a klientem.
3. **Zespół deweloperski** — zespół odpowiedzialny za implementacje aplikacji. Przeważnie nie wyróżniamy tutaj żadnego lidera, a więc struktura zespołu jest płaska:

* programiści
* tester – współpracuje z klientem przy pisaniu scenariuszy i uruchamia testy sprawdzające realizacje scenariuszy
* śledzący (tracker) – sprawdza postęp prac oraz weryfikuje wszelkie oszacowania
* Dokumentacja (przy 4-osobowym zespole min 1 osoba i druga do pomocy)

Programowanie ekstremalne:

1. programiści
2. tester – współpracuje z klientem przy pisaniu scenariuszy i uruchamia testy sprawdzające realizacje scenariuszy
3. konsultant (okazjonalnie - wykładowca)
4. Dokumentacja (przy 4-osobowym zespole min 1 osoba i druga do pomocy)
   1. Metodologia pracy

**Scrum**

Proces realizacji projektu z wykorzystaniem metodyki Scrum można w uproszczeniu przedstawić następująco:

* **programiści**
* **tester** – współpracuje z klientem przy pisaniu scenariuszy i uruchamia testy sprawdzające realizacje scenariuszy
* **śledzący (tracker)** – sprawdza postęp prac oraz weryfikuje wszelkie oszacowania
* **trener (coach)** – nadzoruje proces tworzenia oprogramowania przez zespół, pomaga, kontroluje, itp.
* **konsultant (okazjonalnie)**

W scrumie mamy podział na 3 główne role:

**Scrum Master** — osoba czuwająca nad tym, żeby były przestrzegane wszystkie zasady Scruma. Do jej obowiązków należy również pomoc w rozwiązywaniu problemów organizacyjnych.

**Product Owner** — osoba reprezentująca klienta i dbająca o całokształt projektu. Bardzo często jest ona członkiem zespołu i jest pośrednikiem pomiędzy zespołem deweloperskim a klientem.

**Zespół deweloperski** — zespół odpowiedzialny za implementacje aplikacji. Przeważnie nie wyróżniamy tutaj żadnego lidera, a więc struktura zespołu jest płaska.

Plusy:

* Wykładowca się jej spodziewa, ponieważ na takim pracują w firmie i będzie miało to niewątpliwie plus jeśli chodzi o ocenianie oraz możliwe zatrudnienie (tak insynuuje każda jego wypowiedź na temat projektu)
* Scrum zawiera w sobie cechy metodyk zwinnych. Jest to najbardziej znane podejście do tworzenia oprogramowania i na pewno prędzej czy później się z nim spotkacie.
* Możliwość bieżącego nanoszenia zmian w projekcie w czasie jego realizacji - zakres projektu jest otwarty przez cały okres jego realizacji. Klient może więc nanosić zmiany zarówno w obrębie istniejących, jak i przyszłych funkcjonalności.
* Możliwość bieżącej regulacji zakresu projektu i zamknięcia go w dowolnym momencie - jeśli na skutek np. okrojonego budżetu projektu chcielibyśmy zrezygnować z realizacji części funkcjonalności, jest to jak najbardziej możliwe. To klient decyduje który sprint kończy dany projekt.
* Bieżący monitoring i ocena powstającego produktu - po każdym sprincie klient otrzymuje nową wersję produktu, którą może przetestować i nanieść swoje korekty, możliwe do zrealizowania w kolejnym przyroście.
* Formalności i biurokracja zmniejszone do minimum - dokumentacja projektowa nie jest niezbędna (wystarczą wymagania klienta i bieżący kontakt z nim), nie musimy więc ponosić kosztów jej wytworzenia.
* Ponoszenie wydatków wyłącznie za wykonaną pracę - faktyczny czas realizacji zagadnień przez firmę wdrożeniową może się nieco różnić od szacowanego. Klient ponosi koszty rzeczywistego czasu realizacji - co działa oczywiście zarówno w przypadku szybszego, jak i bardziej czasochłonnego wykonania zadania.
* Szybsza i skuteczniejsza realizacja projektu - dzięki bieżącym korektom kierunku rozwoju produktu, powstaje on szybciej i jest lepiej dopasowany do potrzeb klienta.

Minusy:

* **Brak wiedzy o dokładnym koszcie realizacji projektu** - bazujemy na zarysie funkcjonalnym projektu i szacunkowym koszcie realizacji. Rzeczywisty koszt zależy od końcowych założeń i faktycznym czasie ich realizacji przez firmę wdrożeniową.
* **Ponoszenie wszelkich kosztów związanych z projektem** - jeśli firmie wdrożeniowej z jakichś przyczyn nie uda się zrealizować zagadnienia w szacowanym czasie, rzeczywisty koszt stworzenia funkcjonalności rośnie - i my, jako klient musimy go pokryć. Oczywiście, tak jak sygnalizowałem wyżej, działa to w obie strony.
* **Ryzyko rozrastania się projektu** - mając możliwość zmian i dokładania kolejnych funkcjonalności w trakcie realizacji projektu, bez należytej kontroli, możemy doprowadzić do jego niechcianego i kosztownego rozrastania się.
* **Ryzyko przekroczenia zakładanego budżetu** - punkt będący konsekwencją powyższych. W przypadku niedostatecznej kontroli nad projektem, wymaganiami i jego budżetem, zakładany budżet szybko się wyczerpie, a produkt nie będzie realizował kluczowych funkcjonalności.

**Programowanie ekstremalne**

* **Brak planowania z góry** — nie ustalamy na sztywno żadnej konkretnej architektury systemu. Oczywiście przed rozpoczęciem planu definiujemy jakieś wzorce, jednak przyjmujemy do wiadomości, że architektura ulegnie zmianie w ramach rozwijania projektu.
* **Ciągła modyfikacja architektury** — ten punkt jest związany z poprzednim. Chodzi o to, aby, nie bać się ruszać architektury, jeśli ma to nam pomóc w dalszym rozwoju projektu. Trzeba pisać też kod w taki sposób, aby takie modyfikacje nie ciągnęły za sobą zmian w większości klas projektu.
* **TDD (programowanie napędzane testami)** — testy jednostkowe (pisałem o nich w odcinku o testowaniu) powinny być pisane jeszcze zanim powstanie kod, który mają testować. Dopiero później piszemy kod, który musi zaliczyć wszystkie te testy. Takie podejście sprawia, że pokrycie kodu testami jest bardzo duże, a dodatkowo wymusza na nas pisania kodu, do którego da się łatwo napisać testy.
* **Programowanie parami** — programiści powinni siedzieć parami. Jeden zajmuje się pisaniem kodu, drugi z nich analizuje cały czas kod oraz zgłasza jakieś poprawki. Co kilkadziesiąt następuje zamiana rolami. Ten punkt jest dość kontrowersyjny. Z jednej strony takie podejście zwiększa czytelność kodu i minimalizuje ryzyko pomyłki. Z drugiej strony potrzebujemy znacznie więcej programistów.
* **Iteracyjność** — pracujemy w cyklach o określonej przez nas długości. Na początku zakładamy, co uda nam się zrobić w danym cyklu. Po upłynięciu czasu wysyłamy klientowi aktualną wersję aplikacji oraz planujemy, co zrobimy w kolejnym cyklu.
* **Kontakt z klientem** — nie da się ustalić wszystkich szczegółów przed rozpoczęciem projektu. Dlatego ważne jest, aby wszelkie wątpliwości rozwiewać poprzez bezpośredni kontakt z klientem.

Od siebie dodam, iż jest to nasz najbardziej ulubiony model pracy. Z ostatnich naszych projektów to wynika. Ciężko nam coś zaplanować z góry, wolimy na bieżąco rozwijać projekt oraz na przestrzeni produkcji wpadamy na nowe pomysły i ulepszenia.

Jednakże nie sądzę żeby taki model przypadł do gustu „GLOBALOWI”, tam stawiają na planowanie i sumienność w wyznaczonych na początku działaniach.

## **Metodyki zwinne**

* Być może spotkaliście się ze słowem Agile, które z angielskiego oznacza zwinny. Jest to też nazwa jednej z popularniejszej metodyki prowadzenia projektu. O jej popularności może świadczyć fakt, że teraz każda firma reklamuje się pracowaniem według zasad agile.
* Cała idea polega na samo organizujących się zespołach. Oznacza to, że przykładowo zespół mobilny sam dobiera sobie odpowiednie narzędzia i wybiera architekturę odpowiednią dla danego projektu. Podobnie jak w programowaniu ekstremalnym, miarą postępów projektu jest działająca wersja aplikacji. Osiągamy to poprzez regularne wydawanie kolejnych wersji programu do klienta.
* Bardzo ważną kwestią jest także ciągła poprawa. Zespoły powinny aranżować spotkania, na których poddadzą analizie ostatnio wykonane postępy. Na podstawie wyciągniętych wniosków, wszyscy powinni wspólnie pomyśleć, co można by poprawić na przyszłość. W metodyce Agile bardzo ważna jest odpowiednia komunikacja. Najlepiej, jeśli odbywa się ona twarzą w twarz, chociaż zdarzają się też zespoły, w których każdy z pracowników pracuje z własnego domu

1. Stworzenie ogólnej Specyfikacji wymaga Systemu:
   1. Analiza wymagań
      1. Funkcjonalne
         1. Funkcje systemu

* Zakładanie kont kierowców.
* Definiowanie pojazdów.
* Przypisywanie kierowców do pojazdów (historia przypisań).
* Raportowanie aktualnej pozycji i prędkości.
* Detekcja przekroczeń prędkości.
* Interfejs Web API dla przeglądarek internetowych i aplikacji mobilnych.
  + - 1. Oferowane usługi
* Przeglądanie danych i wyszukiwanie aktualnej i historycznej pozycji pojazdów.
* Przebieg dzienny auta.
* Najczęściej użyte miejsca postoju.
* Najdłuższa oraz najkrótsza trasa z/do pracy po km oraz czasie.
* Zużycie paliwa miesięczne według dłuższa i krótsza trasa (obliczane z ilości tankowań na miesiąc).
* Podpowiedzi przy wyborze trasy na podstawie danych z najdłuższa i najkrótsza trasa z zeszłego tygodnia tego dnia.
  + - 1. Reakcje na dane wejściowe
* Obliczenia zużycia paliwa,
* Obliczenia tras,
* Obliczenia zużycia paliwa,
* Wnioskowanie którą trasę wybrać,
* Podpowiedzi najekonomiczniejszego kierowny.
  + 1. Niefunkcjonalne

1. Pobieranie danych z gps,
2. Przetwarzanie danych położenie, spalanie, postoje itp.
3. Obliczanie,
4. Zaznaczanie gps miejsc postoju,
5. Wyznaczanie najlepszej trasy,
6. Zbieranie danych o czasie,
7. Pobieranie/Zapis danych z bazy NoSql,
   * 1. Diagram przypadków użycia

* + 1. Model danych

NoSql:

Nierelacyjne bazy danych doskonale wpisują się w trend Big Data, ponieważ – w przeciwieństwie do klasycznych silników – pozwalają na szybką analizę danych niestrukturyzowanych i badanie korelacji pomiędzy nimi. W tradycyjnej bazie schemat i relacje są w narzucone z góry a za pomocą odpowiednich zapytań SQL możemy uzyskać strukturalne odpowiedzi mieszczące się we wcześniej opisanych ramach. Przykładowo w banku możemy znaleźć dane o kliencie np. w systemie kredytowym i CRM jeśli tabelę z obu powiążemy np. za pomocą pola PESEL. Dzięki relacji między tabelami będzie można otrzymać informacje, ile osób z danej ulicy – czy danego miasta – posiada kredyt większy niż 100 tys. zł. Będzie to możliwe, ponieważ tabelę zawierającą nazwę ulicy w systemie CRM z tabelą z systemu kredytowego łączy właśnie – nawet jeśli nie bezpośrednio – informacja o numerze ewidencyjnym PESEL.

5 typów baz danych nierelacyjnych:

* **Column:** Accumulo, Cassandra, Druid, HBase, Vertica, SAP HANA
* **Document:** Apache CouchDB, ArangoDB, Clusterpoint, Couchbase, DocumentDB, HyperDex, IBM Domino, MarkLogic, MongoDB, OrientDB, Qizx, RethinkDB
* **Key-value:** Aerospike, ArangoDB, Couchbase, Dynamo, FairCom c-treeACE, FoundationDB, HyperDex, InfinityDB, MemcacheDB, MUMPS, Oracle NoSQL Database, OrientDB, Redis, Riak, Berkeley DB
* **Graph:** AllegroGraph, ArangoDB, InfiniteGraph, Apache Giraph, MarkLogic, Neo4J, OrientDB, Virtuoso, Stardog
* **Multi-model:** Alchemy Database, ArangoDB, CortexDB, Couchbase, FoundationDB, InfinityDB, MarkLogic, OrientDB