VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

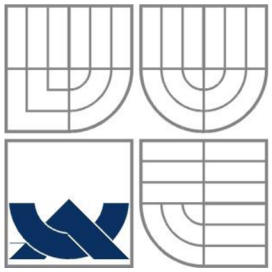
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

WEBOVÁ SLUŽBA PRO ANALÝZU VÝSLEDKŮ MĚŘENÍ POLOVODIČOVÝCH SOUČÁSTEK

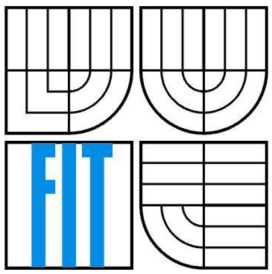
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Jan Střálka
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

WEBOVÁ SLUŽBA PRO ANALÝZU VÝSLEDKŮ MĚŘENÍ POLOVODIČOVÝCH SOUČÁSTEK

WEB SERVICE FOR ANALYSIS OF MEASUREMENT RESULTS OF SEMICONDUCTOR CIRCUIT ELEMENTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

Jan Střálka

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Radek Kočí, PhD

BRNO 2013

Abstrakt

Cílem této bakalářské práce je navrhnout a implementovat softwarový systém pro zvýšení výstupní kvality distribuovaných čipů pomocí vyřazování potenciálně vadných čipů. Projekt je zpracován pro firmu ON Semiconductor, výrobce polovodičů v Rožnově pod Radhoštěm. Software je psaný v Javě a architektura je založena na modulech s algoritmy, které se aplikují na mapy výsledků měření polovodičových součástek (wafer mapy). Zpracovávání běží na aplikačním serveru Tomcat, aplikace je řízena přes rozhraní webové služby.

Abstract

The aim of the bachelor thesis is to design and implement software system for increase the output quality of the distributed chips by phasing out potentially defective chips. This project is designed for ON Semiconductor, manufacturer of semiconductors in Rožnov pod Radhoštěm. The software is written in Java and the architecture is based on the modules with algorithms that are applied to measurements map of semiconductor devices (wafer maps). The system runs on Tomcat application server, and the application is managed via web service interface.

Klíčová slova

Java, web service, polovodiče, defektivita čipů

Keywords

Java, web service, semiconductor, chip defectivity

Citace

Střálka Jan: Webová služba pro analýzu výsledků měření polovodičových součástek, bakalářská práce, Brno, FIT VUT v Brně, 2013

WEBOVÁ SLUŽBA PRO ANALÝZU VÝSLEDKŮ MĚŘENÍ POLOVODIČOVÝCH SOUČÁSTEK

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Radka Kočího, PhD.

Další informace mi poskytl Ing. Vít Matějka (externí vedoucí ze společnosti ON Semiconductor).

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Střálka
Datum 15. 5. 2013

Poděkování

Chtěl bych vyjádřit poděkování za vstřícný přístup a odbornou pomoc, jak svému vedoucímu Ing. Radkovi Kočímu, PhD., tak externímu vedoucímu Ing. Vítu Matějkovi. Díky patří i Ing. Jaroslavovi Bernkopfovi a dalším kolegům z ON Semiconductor za cenné rady a pomoc při shánění odborných materiálů.

Obsah

1 Úvod

Cílem tohoto projektu je zvýšit výstupní kvalitu čipů, které jsou distribuovány zákazníkovi, pomocí vyřazování potenciálně vadných čipů. Mým zadáním pro bakalářskou práci je navrhnout a vytvořit softwarový systém, který by to umožňoval. Architektura systému je postavena na modulech s algoritmy, které mají za úkol detekovat a označit potenciálně vadné čipy na mapách výsledků měření polovodičových součástek (wafer mapy). Současně navržené algoritmy jsou založeny na deterministických statistických metodách, ale je možné implementovat libovolné vlastní algoritmy. Software je psaný v Javě na platformě Java EE. Aplikace se skládá ze serverové a klientské části. Zpracovávání běží na serverové části a klientská aplikace s ní komunikuje pomocí webové služby. Jako aplikační server byl vybrán Tomcat.

Projekt je zpracován pro firmu ON Semiconductor, výrobce polovodičů. Nadnárodní koncern se sídlem v americkém Phoenixu má v Rožnově pod Radhoštěm jeden z nejvýznamnějších výrobních závodů. Má zde hlavní část produkce, ale také sdílené centrum IT služeb, které je největší v korporaci. Zabývá se vývojem, nasazováním a podporou informačních systémů pro potřeby výrobních závodů nadnárodní skupiny ON Semiconductor v oblasti řízení výroby, zpracování dat a automatizace. [1]

2 Technologie výroby polovodičů

Ze všeho nejdříve popíši technologii polovodičové výroby. Popis tohoto technologicky náročného procesu, který vyžaduje vysokou přesnost zpracování a čistotu prostředí, je nezbytný pro pochopení vzniku vad v čipech. Bude zde popsána výroba tak, jak probíhá ve firmě ON Semiconductor, ale většina postupů platí pro polovodičové výroby obecně.

2.1 Tažení monokrystalu

Základem pro výrobu polovodičů je křemíkový monokrystal. Ten je tažen v pecích, kde je do taveniny vložen zárodečný krystal z vysoce čistého křemíku. Zde musí být udržována přesná teplota a eliminovány otřesy okolního prostředí. Výsledný monokrystal (obrázek 1) má tvar válce, pomineme-li horní část a spodní část, které se nevyužívají. Jeho průměr může být až 40 cm (16 palců) a výška až 2 m (v ON Semiconductor je obvyklý průměr 6 palců). Tento polotovar se pak musí ještě opracovat, zvnějšku je obroušen tak, aby jeho půdorys byl přesně kruhový, horní a spodní nevyužívané části jsou uřezány (získá přesný tvar válce).

Obrázek 1 - Křemíkový monokrystal

Později bude nutné identifikovat orientaci. Dosáhne se toho tím, že se kruhový půdorys něčím naruší. Způsoby jsou dva, tím prvním je tzv. fazeta, která spočívá v uříznutí malé části jedné strany kruhového půdorysu do roviny (používá se u monokrystalů malých průměrů, obrázek 2). Druhým způsobem je do kruhového půdorysu vyfrézovat trojúhelníkovou rysku, které se říká notch (používá se u monokrystalů větších průměrů, vyřadí se menší objem materiálu).

2.2 Vytváření waferu

Monokrystal je rozřezán diamantovým řezačem na kruhové desky nazývané wafer (česky také jen jako „deska“), vznikne tím tzv. základní substrát pro vytváření čipů. Běžná tloušťka waferu je zhruba 0,5 mm. Obecně platí, že čím je wafer tenčí, tím méně materiálu se použije na jeden čip a tím jsou nižší výrobní náklady. Ušetřit se dá i na průměru, je snaha dělat wafery velkých průměrů, protože se při stejném počtu výrobních operací vyrobí více čipů. Vyšší jsou ale vstupní náklady související s technologickou náročností, protože větší monokrystal je křehčí.

Tyto desky jsou broušeny tak, že získají zrcadlový lesk. Po obroušení se ještě deska ponoří do slabé kyseliny, která má vyhladit ty nejjemnější nerovnosti. Kyselina se poté smyje deionizovanou vodou a deska se osuší.

Výsledná deska je již hotovým substrátem připraveným na výrobu čipů. Tyto wafery jsou pak distribuovány do výrobny čipů. [2]

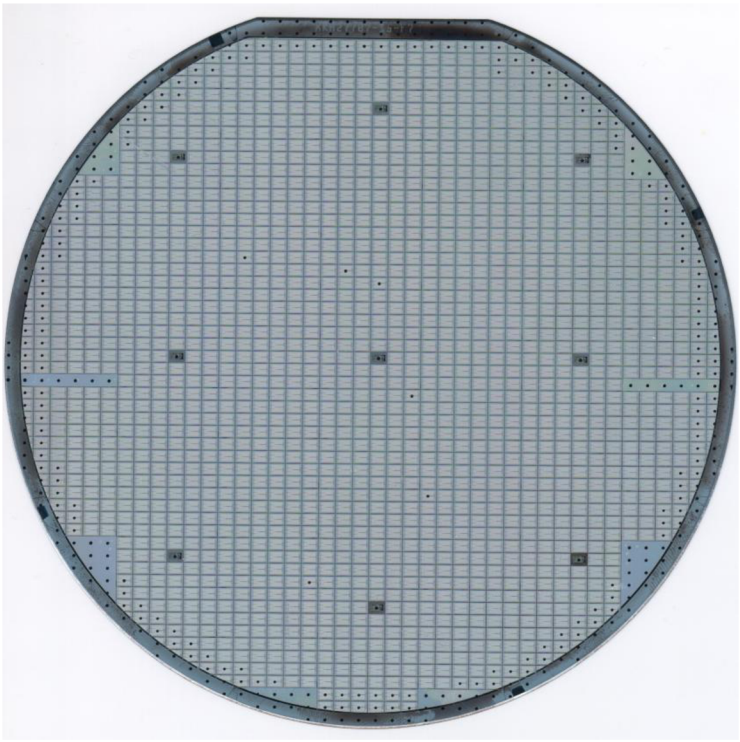
2.3 Výroba čipů

Wafery se nejprve zařadí do výrobních várek, neboli sad (lot), které čítají až 50 kusů. Je to důležité i z logistického hlediska, protože každá deska je pak dohledatelná pomocí identifikátoru sady a čísla desky.

Záměrem je na desce vytvořit přesně definované oblasti, které jsou řízeně dotovány příměsmi, pro dosažení polovodiče typu N či P. Pro polovodič typu N se používá fosfor, arsen a antimon. Této příměsi se říká donor, protože jejích pět valenčních elektronů ve struktuře „daruje“ jeden volný elektron. Naopak pro získání polovodiče typu P se používá bór a hliník, prvky se třemi valenčními elektrony. Ve struktuře pak chybí jeden elektron.

Tyto oblasti mohou být na každé desce v několika vrstvách, přičemž každá vrstva má jinou mapu oblastí (místa, kam jsou cíleny příměsi). Vytvoření každé vrstvy představuje sekvenci.

Sekvence se skládá z nanesení povrchu desky materiálem, který je pro příměsi nepropustný. Používá se například asi 500 nm tlustá oxidační vrstva. Pomocí fotomasky, která se též nazývá retikl, se tato plocha rozdělí na žádoucí oblasti. V těchto oblastech se odleptá oxid až na křemíkový povrch. Na desku se znovu nanese vrstva oxidu, tentokrát tlustá jen 20 nm, aby přes ni pronikla příměs. Poté je implantována, deska se zahřeje na vysokou teplotu (850 až 1200 ℃), aby došlo k difuzi příměsi do materiálu.



Obrázek 2 - Wafer s již vyrobenými čipy

Tento proces se opakuje, dokud nejsou naneseny všechny vrstvy. Je třeba poznamenat, že s každou další vrstvou a následným zahřáním se předchozí vrstvy difundují hlouběji do desky a povrch jednotlivých oblastí se rozšiřuje. Můžou se vyskytovat i další typy vrstev, například různé kovy na propojení jednotlivých součástek. Mezi vrstvami se vždy deska očistí. Poslední vrstvou je izolační nitrid (Si_3N_4), který odizoluje funkční vrstvy od okolního prostředí. Tato bariérová vrstva je odleptána jen na vybraných místech, kam patří kontaktní plošky, které propojují polovodičovou strukturu s okolním prostředím. [2]

Na obrázku 2 je wafer s již vyrobenými čipy (poznáme podle zřetelné mřížky, která ohraničuje čipy).

2.4 Testování čipů

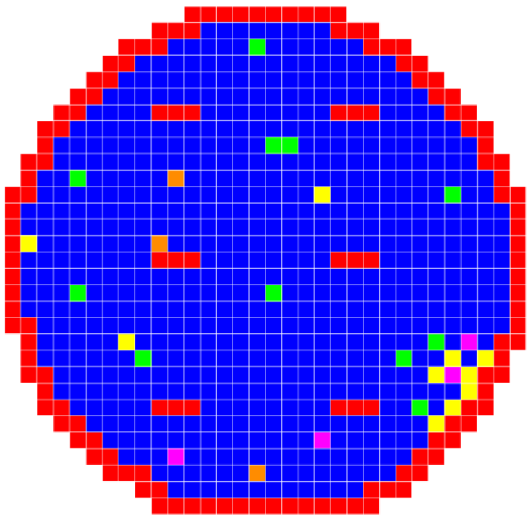
Po vyrobení čipů jsou desky kontrolovány zařízením zvaným tester. Tester provádí elektrická měření na čipech a vyhodnotí vadný čip včetně typu naměřené chyby. Princip měření a typy testů přesahují rozsah této práce, ale podstatné je, že pro různé typy čipů (konkrétní model zesilovače, hradla NAND, apod.) existuje různá množina prováděných testů. Tester generuje dva výstupy. Naměřené data (např. konkrétní hodnoty napětí a proudů) jsou uloženy jako parametrická data do STDF souboru. Vyhodnocení testu pro každý čip se zanese do tzv. binu.

Bin je celočíselná hodnota, pokud je nulová, indikuje správný čip, v jiném případě znamená, že je čip vadný nebo se z nějakého jiného důvodu nevyužije. Pro každý typ čipu existuje tabulka, která mapuje číslo binu na jeho význam.

2.5 Wafer mapa

Od naměření testerem se dostáváme k předělu mezi fyzickým waferem a logickou mapou zpracovatelnou počítačem. Základ wafer mapy tvoří dvourozměrné pole, hodnotami jsou čísla binů. Grafické zobrazení mapy vidíme na obrázku 3Obrázek 3, kde jsou čísla binů zobrazeny jako barvy, přičemž modrá značí dobré čipy, červená značí okrajové a testovací čipy, a jiné barvy představují vadné čipy. Kromě binů obsahuje další informace o mapě týkající se výrobního procesu a pozdějšího zpracování. Základními informacemi jsou identifikátor sady (lot id), číslo desky (wafer id) a identifikátor mapy (map id).

Ze získané wafer mapy lze vypočítat výtěžnost (yield), což je poměr dobrých čipů vůči všem potenciálně použitelným čipům. Ideální výtěžnost je 100 %, ale v praxi bývá nižší. Odchylky ve výtěžnosti oproti normálu mohou indikovat systematickou chybu ve výrobním procesu.



Obrázek 3 - Wafer mapa

2.6 Dodatečné zpracování

Protože elektrické testy prováděné testerem v některých případech nepodchytí čipy ohrožené skrytými vadami, provádí se dodatečné zpracování. V praxi dochází hlavně k tomu, že čip projde testem, dokonce může být i funkční, ale jeho životnost je kratší, než by měla být. Z hlediska požadavků výstupní kvality je nutno brát tyto čipy také jako vadné. Otázkou je, které čipy z těchto správně vyhodnocených jsou opravdu dobré a které jsou vadné. Proto necháváme wafer mapu projít dodatečným zpracováním. Toto se provádí analýzou pouze doposud naměřených dat. Hlavním problémem je tyto čipy odhalit a tento problém je základem pro mou práci. Systém, jehož úkolem je toto řešit a který mám navrhnout, jsme interně nazvali *Post Processing*.

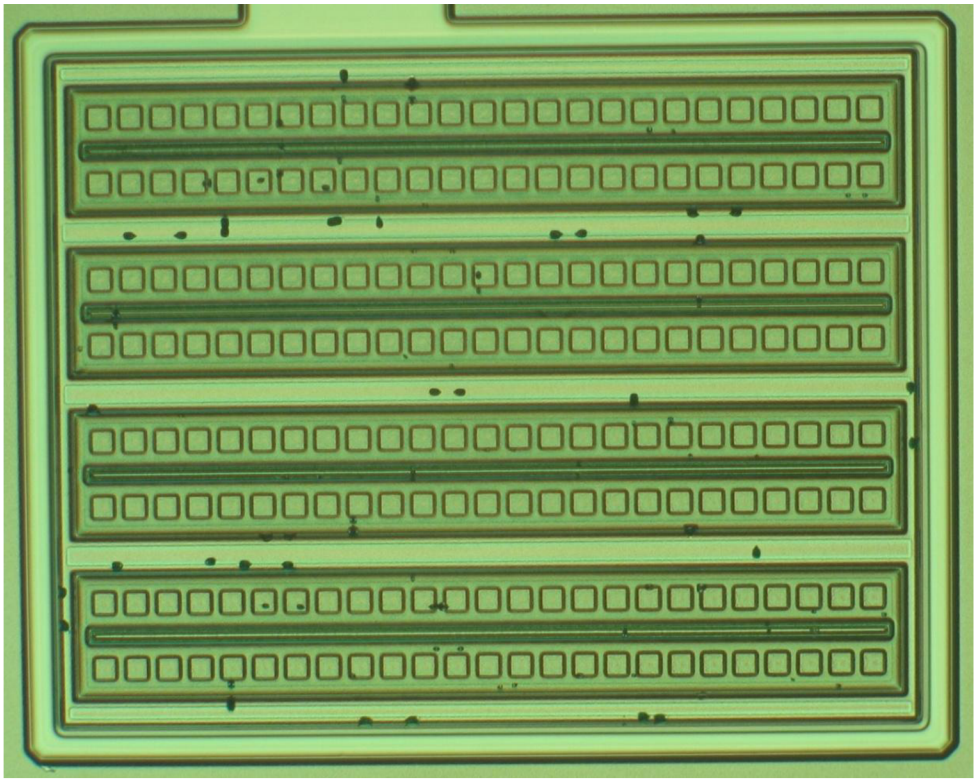
Konkrétně má systém za úkol analýzu stávajících dat takovým způsobem, že využívá určité systematičnosti při výskytu těchto potenciálních vad. Zjednodušeně řečeno se statisticky zpřísňují kritéria. Zpřísňujícími kritérii jsou v případě tohoto systému algoritmy, které vyhledávají určité vzory nebo pravidelnosti a podle nich vyhodnocují další potenciálně vadné čipy. Tím se sice snižuje výtěžnost, ale zvyšuje kvalita čipů, které jsou distribuovány zákazníkovi. Je klíčové vhodně volit zpřísňující kritéria, tak aby se zbytečně nevyřazovaly dobré čipy. Algoritmům pro odhalování potenciálně vadných čipů se věnuje kapitola 3.

2.7 Vady a jejich analýza

Vady mohou být způsobeny různými příčinami. Některé vady jsou přirozené, jiné mohou vzniknout chybou ve výrobním procesu. Těmto vadám se dá většinou systematickým přístupem vyvarovat. Je ale otázkou, za cenu jakého úsilí. Ať už vzniknou lidským faktorem či odchylkou výrobního zařízení, přijměme fakt, že se stávají a mým úkolem je pouze eliminovat důsledky.

2.7.1 Prach a jiné nečistoty na povrchu

I přes eliminaci prachu v čistých prostorách, kde je filtrovaný vzduch a lidé se zde pohybují pouze v čistých kombinézách, se občas vyskytne nějaká částice prachu, která dopadne na povrch desky. Vlivem zahřívání v peci se může nečistota vtavit pod povrch a narušit tím strukturu čipu. Nebo se také vtavit nemusí, ale překáží při implantaci příměsí (maskování implantace). Na obrázku 4 vidíme tento neduh na zvětšeném čipu pod mikroskopem. Že se může jednat o tuto příčinu, lze z wafer mapy vyčíst tak, že vadné čipy jsou rozmístěny nesystematicky. Pokud vadné čipy tvoří shluky, je nezbytné ošetřit toto okolí. Samostatné vadné čipy obvykle netřeba ošetřovat.



Obrázek 4 - Částice prachu na čipu

2.7.2 Pnutí mezi různými vrstvami na desce

Jednotlivé nanášené vrstvy mohou být z různých materiálů, které mají odlišnou tepelnou roztažnost. Vlivem zahřívání v peci nebo při chladnutí se pak může stát, že spojení mezi vrstvami popraská a například se vytvoří svody, přeruší se kontakt nebo se čip bude chovat nestandardním způsobem. Obvykle vady tohoto typu tvoří na desce souvislé oblasti.

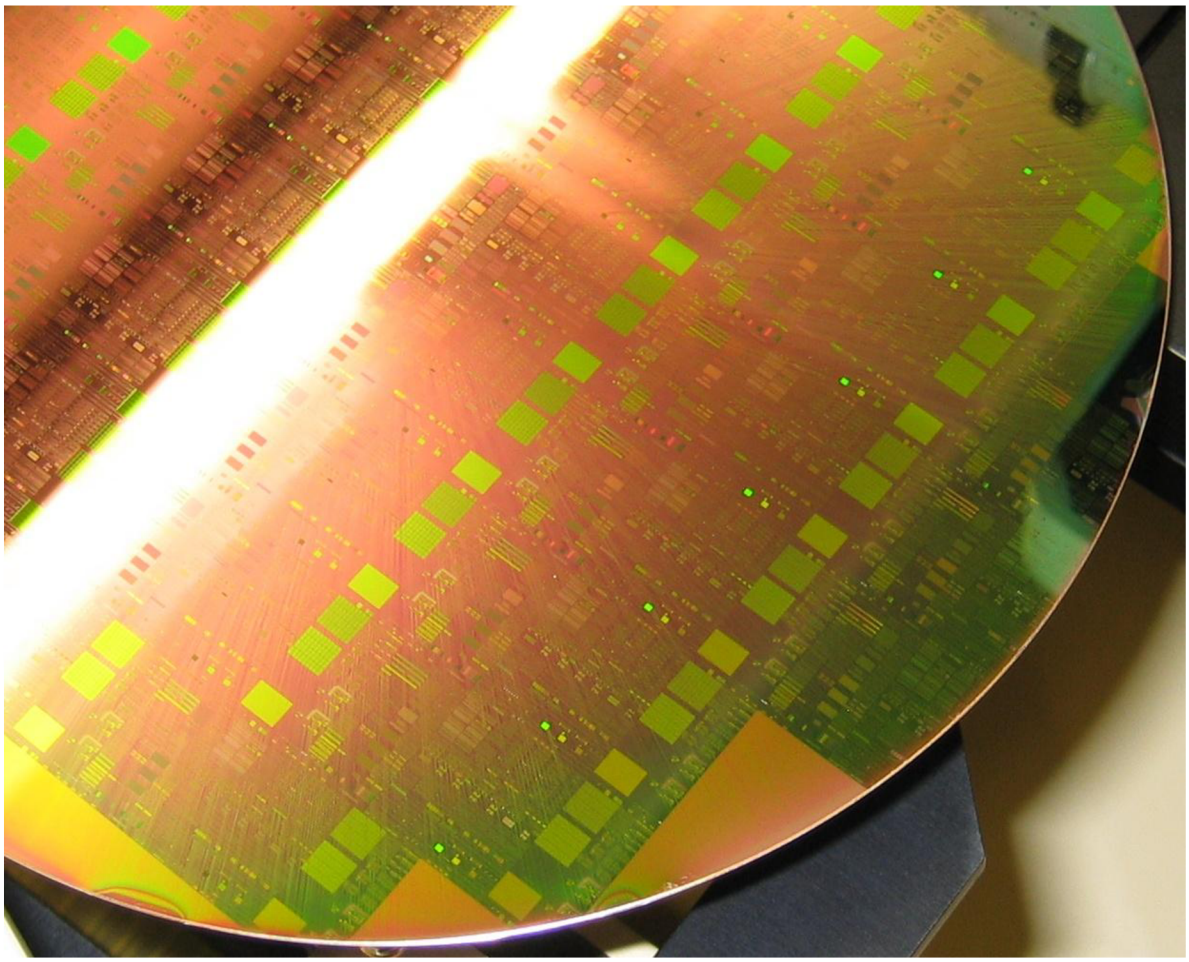
2.7.3 Vadný retikl

Retikl je obdélníková fotomaska, která nepokrývá celou plochu waferu, ale je na každý wafer nanášena opakovaně. Jednotlivé pozice retiklu pak tvoří na waferu pomyslnou mřížku. Dělá se to tak z toho důvodu, že světlo (vycházející z jednoho bodu) dopadající na malý retikl má na okraji retiklu menší úhel než u velkého retiklu a dosáhneme tím větší přesnosti.

Pokud existuje nějaká vada či nečistota na retiklu, pak je vysoce pravděpodobné, že se chyba přenese na čipy, které jsou pod ní. Poznávacím znamením jsou vadné čipy tvořící pravidelnou mřížku odpovídající velikosti retiklu. Jedná se o typicky systematickou chybu a lze ji odhalit rozpoznáním pravidelnosti na pozicích retiklu.

2.7.4 Špatné či nerovnoměrné nanesení vrstvy

Nesprávná tloušťka oxidační vrstvy může ovlivnit hloubku difuze příměsí. To může změnit parametry vyrobeného čipu nebo ho učinit nefunkčním. Nerovnoměrnost může být způsobena například nanášením fotorezistivní vrstvy po kapkách. Kapka je nanesena na otáčející se wafer, kde by se měla vytvořit rovnoměrná vrstva. Pokud má však kapka jinou konzistenci, například je částečně zaschlá, může se po waferu kutálet. Jev se dá poznat pouhým pohledem na dobře osvětlený wafer, kde ze středu waferu vycházejí mikroskopické vroubky připomínající paprsky (obrázek 5). Dalším příznakem je velká hustota vadných čipů, které se vyskytují na okraji.



Obrázek 5 - Paprsky vycházející ze středu waferu indikující špatné nanesení oxidačního materiálu

2.7.5 Vada na měřicích hrotech

Tester se kontaktuje s čipy pomocí měřicích hrotů. Pro urychlení měření je na testeru více skupin hrotů a testuje se několik čipů najednou. Sníží se tím počet posuvů waferu. Hlava testovacího zařízení, které umožňuje změřit více čipů najednou, se nazývá multisite a tvoří obdelník.

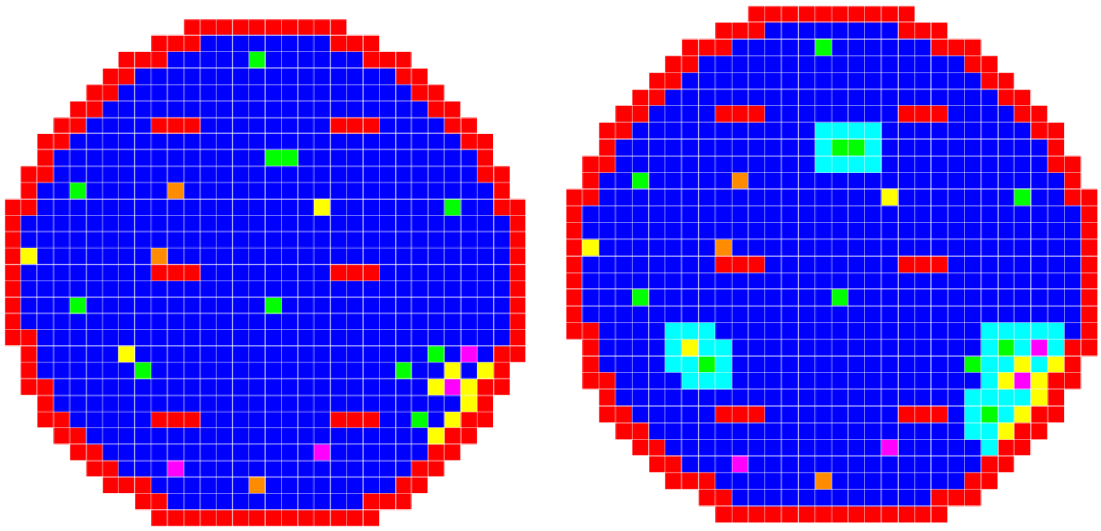
Pokud je v nějaké části multisite poškozen (např. koroze na hrotu), čipy jsou pravidelně vyhodnocovány jako vadné na souřadnicích o násobku velikosti multisite. Je to systematická chyba, která se projevuje velice podobně jako vadný retikl.

3 Algoritmy pro zpracování map

Existuje několik druhů algoritmů na statistické zpracování dat. Všechny algoritmy pracují s wafer mapou, kde je každý čip označen číselným binem. Pomocí parametru je předán seznam, který určuje, jaké biny představují vadné čipy a jaké biny představují dobré čipy.

3.1 Neighbourhood analysis

Tento algoritmus zkoumá čipy v blízkosti daného vadného čipu a podle polohy těchto čipů vůči tomuto čipu i dalším vadným čipům v okolí i podle druhů vad rozhoduje, které z těchto čipů budou prohlášeny také za vadné. Konkrétní rozhodovací kritéria se mohou lišit podle typu čipu, technologie atd. Příklad aplikace algoritmu vidíme na obrázku 6.



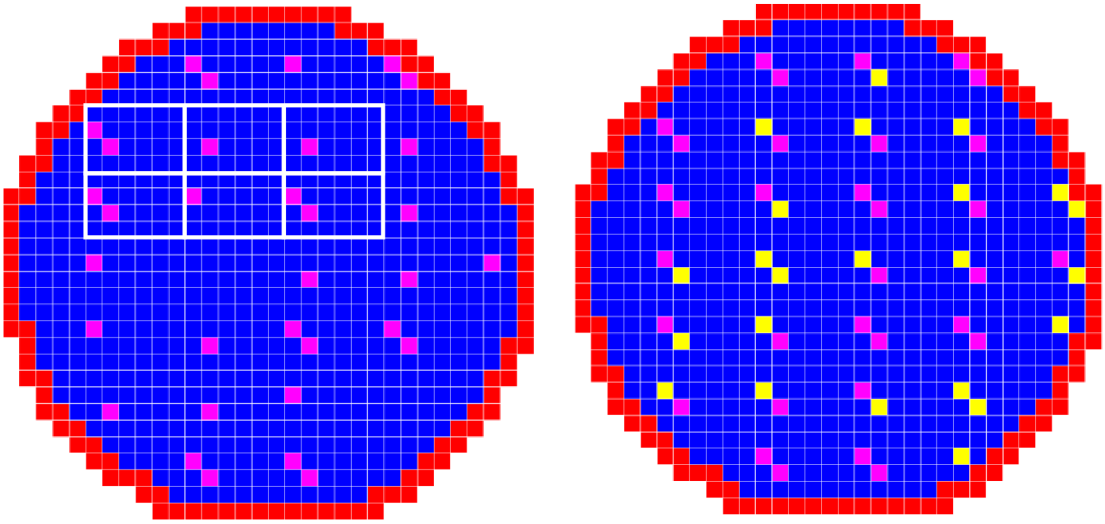
Obrázek 6 - Algoritmus Neighbourhood s minimální velikostí shluku 2
(vlevo před zpracováním, vpravo po zpracování)

Zjistilo se, že pokud se vyskytne shluk vadných čipů, je pravděpodobné, že jsou potenciálně vadné i čipy v blízkém okolí tohoto shluku. Základní varianta algoritmu počítá s tím, že se vyberou čipy v osmi-okolí od všech stávajících vadných čipů. Tyto vybrané čipy se označí za vadné. Dále je možné ovlivnit chování algoritmu pomocí parametrů. Například zdali okrajové čipy na waferu budou brány jako vadné. Nebo mohou být za špatné označeny čipy, které jsou v uzavřeném okruhu vadných čipů. Pseudokód ke zjednodušené variantě *Neigbourhood Analysis* je uveden v příloze 1.A.

3.2 Positional analysis

Jedná se o analýzu výskytu vad čipů způsobených retiklem nebo multisitem. Retikl nebo multisite tvoří obdelník, opakovaně nanášený na pokrytí celé plochy waferu. Proto pokud je původcem vad, tak jsou tyto vady rozmístěny do pravidelné mřížky. Na základě rozpoznání opakujících se vad na stejném místě v rámci retiklu nebo multisite je možno rozhodnout o označení určitých čipů za potenciálně vadné na celém waferu. Prah pravidelnosti (hodnota v procentech) je volitelný.

Příklad vidíme na obrázku 7, kde je na wafer použit retikl 6×4 (pro ilustraci bylo několik retiklů na obrázku zvýrazněno). Vlevo je originální naměřená mapa, na které můžeme pozorovat pravidelně se vyskytující 2 vadné čipy. Občas jsou vadné čipy vynechány, což znamená, že nebyly naměřeny jako vadné. Spustíme-li algoritmus *Positional Analysis* s prahem pravidelnosti 40%, vyjde nám mapa na pravé části obrázku 7.



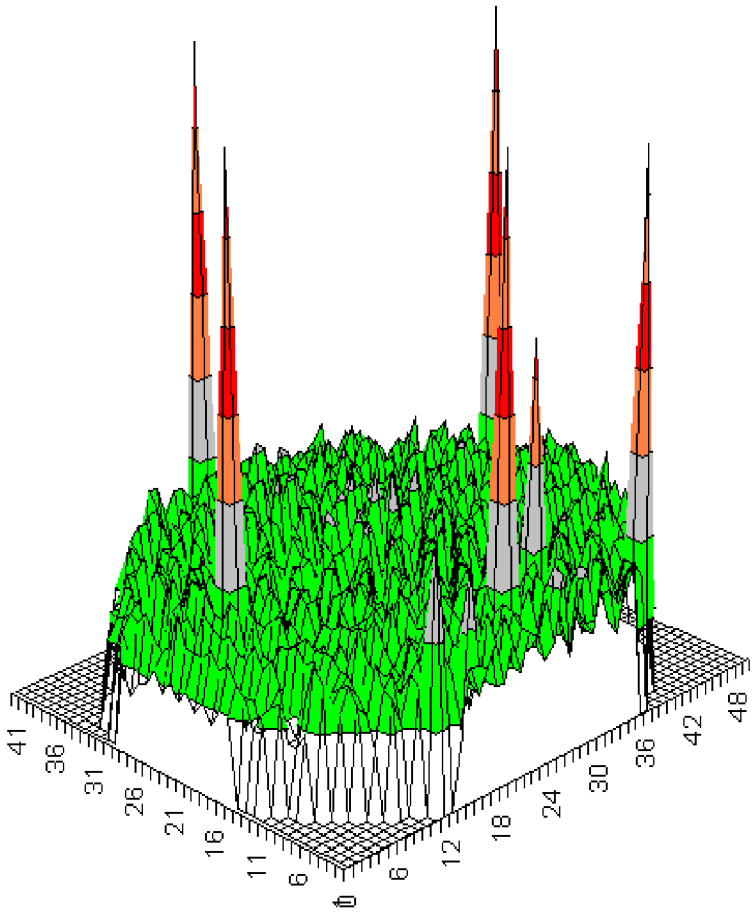
Obrázek 7 - Positional Analysis, vada na retiklu 6×4 (vlevo před zpracováním, vpravo po zpracování)

Pseudokód k algoritmu *Positional Analysis* je uveden v příloze 1.B.

3.3 Dynamic Part Average Testing

Zde se analyzují naměřená parametrická data a zkoumá se, zda vyhovují stanoveným limitům. Zpracovávaná data tentokrát nepředstavují mapu binů, ale mapu naměřených hodnot určité fyzikální veličiny na součástkách na waferu. Mají trojrozměrné zobrazení, příkladem je obrázek 8 (např. jednotlivé hodnoty představují závěrný proud diodou, červené oblasti jsou nečistoty, přes které teče proud). Účelem je statisticky zpřísnit kritéria na základě krátkodobých i dlouhodobých naměřených parametrických dat.

Zdrojem parametrických dat jsou STDF soubory, což je standard pro ukládání výsledků měření a testů polovodičových součástek, používají jej testery.



Obrázek 8 - Parametrická data – ilustrační obrázek

Existují dva typy limitů. Prvním typem je statický limit, což je definované rozmezí, ve kterém se může hodnota pohybovat, a který by měl odpovídat specifikačním normám součástky a vycházet z dlouhodobých naměřených dat. Druhým typem je dynamický limit, jehož interval má střed v mediánu naměřených hodnot. Dynamický limit je zpravidla

podintervalem statického limitu, tudíž přísnějším kritériem. Cílem dynamického rozmezí je nepropustit čipy, které se významně liší od střední hodnoty. Při tomto zpracování jsou nutné dva průchody algoritmem, první průchod vypočítá medián, aritmetický průměr a směrodatnou odchylku, druhý průchod vyřadí čipy podle algoritmu.

3.3.1 Popis algoritmu v bodech

1) **Statický limit I_S** je určen na základě dlouhodobých měření a do algoritmu vstupuje jako konstanta.
2) Prvním průchodem mapou jsou uložena do seznamu naměřená **parametrická data** pro čipy, které byly testerem vyhodnoceny jako dobré, ostatní jsou ignorována. Seznam bude dále nazýván jako **sesbíraná data**.
3) Horní a dolní 3 % sesbíraných dat jsou vyřazeny. Všechny další výpočty jsou provedeny nad zbývajícími 94 % dat (dále jen **vybraná data X**).
4) Je vypočten medián $\tilde{x}$ a aritmetický průměr $\bar{x}$ z vybraných dat X.
5) Je vypočtena směrodatná odchylka σ (rovnice 1).
6) Je vypočten **dynamický limit I_D**, což je interval, jehož střed je stanoven v $\tilde{x}$ a od tohoto středu jsou jeho hranice specifikovány jako +/- σ (rovnice 2).
7) Je vypočten **finální limit I_F**, který je průnikem statického a dynamického limitu (rovnice 3).
8) Druhým průchodem mapou se vyřadí čipy, jejichž naměřené hodnoty nejsou v rozmezí finálního limitu I_F.

$$\sigma = \sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} (x_i - \bar{x})^2}$$

Rovnice 1 - Směrodatná odchylka

$$I_D = \langle \tilde{x} - \sigma; \tilde{x} + \sigma \rangle$$

Rovnice 2 - Dynamický limit

$$I_F = I_S \cap I_D$$

Rovnice 3 - Finální limit

4 Použité technologie

4.1 Webová služba

Webová služba (*Web Service*) je technologie použitelná pro vzdálené volání procedur na serveru. Rozhraní na serveru obsahuje určitou množinu operací a datových typů, se kterými je schopen pracovat. Klientská aplikace má o tomto přehled a může si požádat o seznam operací s jejich parametry a o seznam definic datových typů. Komunikace je založena na protokolu SOAP (*Simple Object Access Protocol*), který je obdobou serializace objektů do XML. Popis rozhraní, tzn. seznam operací a datových typů je popsán ve formátu WSDL (*Web Service Description Language*). [3] [4]

Protokol SOAP není příliš úsporný, ale je dobře čitelný a poměrně jednoduše implementovatelný pomocí zabudovaných knihoven. Typicky se využívá pro klient-server komunikaci mezi aplikacemi psaných v Javě nebo .NET. [5]

V případě tohoto projektu nebudou přes rozhraní webové služby procházet žádné velké datové toky (maximálně jednotky kB), ale pouze řídicí toky pro ovládání průběhu zpracování.

4.2 Java

Java je objektově orientovaný programovací jazyk představený společností Sun Microsystems v roce 1995. Má se za to, že je současně druhým nejpopulárnějším jazykem na světě [6] a je používaný na obrovské škále zařízení, od stolních počítačů, přes výkonné servery, až po mobilní platformy. Jeho syntaxe vychází z C++, ale v mnohém byla zjednodušena. Je multiplatformní, protože je interpretován pomocí virtuálního stroje *Java Virtual Machine*. Správa paměti je automatizována garbage collectorem. [6]

4.3 JAX-WS

Celým názvem je *Java API for XML Web Services*. Toto aplikační rozhraní je součástí platformy Java EE a umožňuje jednoduchou práci s webovou službou. Využívá anotací Javy. [7]

Mezi základní anotace pro vytvoření webové služby patří:

- `@WebService` – umísťuje se před třídu, která má sloužit jako rozhraní WS
- `@WebMethod` – umísťuje se před metodu přístupnou přes rozhraní WS
- `@WebParam` – umísťuje se před parametr WS metody

4.4 JavaServer Pages

Umožňuje vytvářet dynamicky generované webové stránky pomocí jazyka Java. V principu patří tato technologie do stejné kategorie jako široce proslulé PHP. Využívá Java servletů, JSP je pouze jejich vysokoúrovňová abstrakce. Do kódu stránky se vkládá Java kód v blocích, které začínají značkou `<%` a končí značkou `%>`. JSP se zpracuje na aplikačním serveru (např. Tomcat) a klientský webový prohlížeč dostane již zpracovaný kód. [8]

4.5 JavaScript

Tento jazyk nemá s Javou tolik společného, jak by se mohlo dle jeho názvu zdát. Je to skriptovací jazyk, který je interpretován webovým prohlížečem. Na rozdíl od JSP se tedy jeho kód musí přenést po síti až ke klientovi. Jeho syntaxe vychází z jazyků C++ a Java, je objektově orientovaný a používá dynamické přiřazování typů. Jazyk vyvinul *Brendan Eich* a poprvé byl použit v roce 1995 v prohlížeči *Netscape Navigator 2.0*. Dnes je jeho podpora součástí všech běžně dostupných prohlížečů. [9]

Jeho součástí je *Document Object Model*, který umožňuje objektově přistupovat k XML/HTML elementům, interaktivně měnit obsah stránky a reagovat na události.

4.6 AJAX

Zkratka pochází z Asynchronous JavaScript and XML, přičemž se nejedná o samostatný jazyk či technologii, ale o metodiku. Používá se pro provádění dynamických změn malých částí webové stránky bez nutnosti načítat a zpracovávat obsah celé stránky. Vyžaduje podporu klientského skriptování v prohlížeči, nejčastěji JavaScriptu. Skript načte data na pozadí např. ve formátu XML nebo JSON, a vloží je do obsahu již načtené stránky v prohlížeči. [10]

4.7 Apache Tomcat

Tomcat je aplikační server pro Javu. Poskytuje přístup k webovým aplikacím v JSP a Java servletům. Projekt založil James Duncan Davidson, vývojář ze společnosti Sun Microsystem. Později se projekt transformoval do podoby open source projektu a byl darován do rukou Apache Software Foundation. [11]

Skládá se z těchto základních komponent:

- **Catalina** – kontejner pro servlety a JSP
- **Coyote** – konektor pro HTTP, obstarává spojení s klientem
- **Jasper** – engine pro JSP, parsuje JSP do java kódu a servletů

4.8 Apache Maven

Slouží pro správu závislostí mezi projekty. Pokud například projekt *A* používá třídy z projektu *B*, pak při překladu projektu *A* zajistí stažení a přeložení projektu *B*. Typicky se používá pro Javu, ale zvládá i např. C# nebo Ruby. Konfiguruje se tzv. souborem POM (*Project Object Model*), což je XML soubor, ve kterém je popsáno, jakým způsobem má být projekt sestaven a závislosti na externích modulech a komponentách. Při sestavování dokáže tyto závislosti dynamicky stáhnout. [12]

4.9 Mercurial

Mercurial je systém pro správu verzí. Představil ho Matt Mackal v roce 2005. Je to robustní a decentralizovaný systém, který se řadí v současnosti k těm, které nabízí nejširší možnosti. Ovládá se přes konzolovou utilitu `hg`, jednoduše pojmenované podle chemické značky rtuti (mercury). Pro ovládání pomocí GUI existuje projekt *TortoiseHG*. [13]

4.10 NetBeans

NetBeans je multiplatformní integrované vývojové prostředí určené zejména pro Javu, ale i částečně pro PHP či C++. Zajímavostí je, že toto současně nejpoužívanější prostředí pro vývoj v Javě, vychází ze studentského projektu Univerzity Karlovy z roku 1996. V roce 1999 projekt koupila společnost Sun Microsystems, která jej později prodala společnosti Oracle. [14]

5 Návrh aplikace

Zadáním bylo navrhnout a implementovat systém pro dodatečné zpracování wafer map, nazvaný *Post Processing*. Zpracování by mělo běžet na serveru a má být ovladatelné pomocí webové služby (WS). Algoritmy, kterými se budou mapy zpracovávat, nejsou pevně dané a měly by jít snadno přidávat a měnit. Na webovou službu se později plánují připojit další aplikace v infrastruktuře, proto by mělo být rozhraní jednoduše použitelné.

5.1 Souhrn požadavků

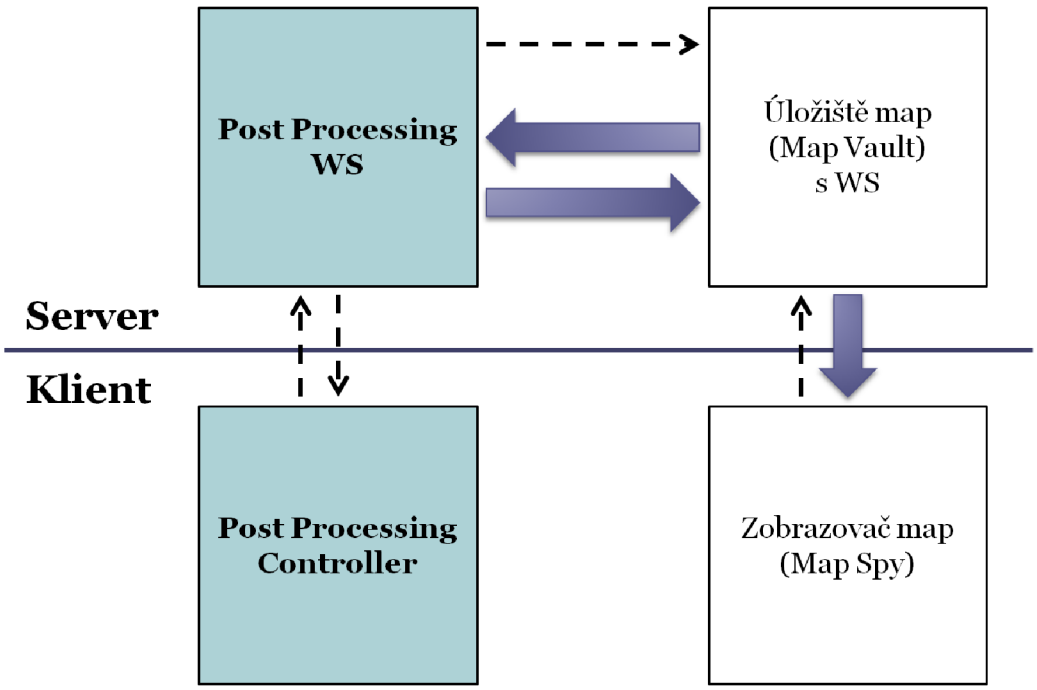
Pro výsledný informační systém byly sestaveny tyto požadavky:

1) Serverová aplikace – *Post Processing WS*
   - Webová služba, která bude poskytovat klientským aplikacím jednotný a kontrolovaný přístup k procedurám pro zpracování dat.
   - Přístup k odpovídajícím úložištím dat, odkud budou stahovány a kam budou ukládány wafer mapy.
   - Mechanizmus asynchronního zpracování klientských požadavků (zpracování může být časově náročné) a poskytování online informací o aktuálním stavu zpracování.
   - Implementace algoritmů ve formě nezávislých modulů.
2) Klientská aplikace – *Post Processing Controller*
   - Jednoduchá webová aplikace na manuální obsluhu serveru.

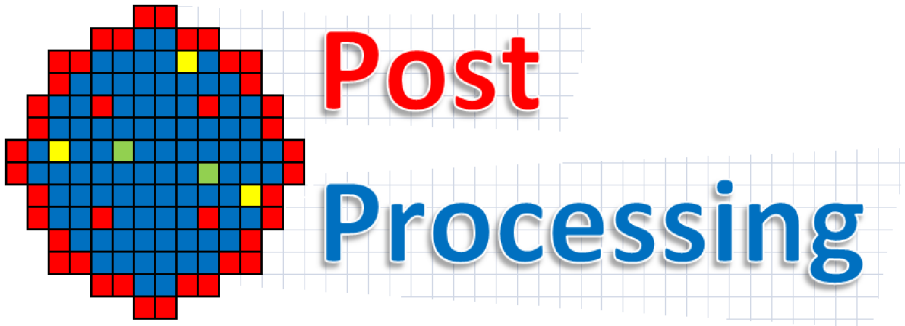
5.2 Blokové schéma systému

Znázornění architektury ve firemní infrastruktuře vidíme na obrázku 9. Šipky propojující jednotlivé boxy představují datové toky. Tenké přerušované čáry jsou řídicí a kontrolní toky (malé objemy dat), tlusté šipky reprezentují datové přenosy wafer map (větší objemy dat). Mým úkolem je navrhnout a implementovat boxy označené modře, tzn. *Post Processing WS* a *Post Processing Controller*. Systém musí spolupracovat také s jinými, již existujícími, prvky ve firemní infrastruktuře. Wafer mapy jsou načítány a ukládány v úložišti *Map Vault*, které má

rozhraní webové služby. Mapy zde mohou být uloženy v různých formátech, pracovat budu s formáty WMXML a INF. Pro komplexní zobrazování map existuje aplikace *UMR Map Spy*. Umožňuje grafické prohlížení map, výpis detailů o mapě. Později se plánuje rozšířit o možnost zadávat požadavky do *Post Processing WS* (čili bude fungovat jako klient pro serverovou aplikaci). Plnohodnotné manuální ovládání serverové aplikace je možné přes webovou aplikaci *Post Processing Controller*, která je jednoduchým klientem.



Obrázek 9 - Znázornění aplikace ve firemní infrastruktuře blokovým schématem



Obrázek 10 – Navrhnuté logo systému Post Processing

6 Implementace serverové aplikace

Serverová aplikace *Post Processing WS* má za úkol zpracovávat wafer mapy pomocí zadaných algoritmů. Zdrojovým i cílovým úložištěm je *Map Vault*. Požadavky na zpracování, stejně jako další operace se serverem, zadávají klienti pomocí webové služby. Z každého přijatého požadavku na zpracování vznikne tzv. úloha, která běží na pozadí a postupně zpracovává požadované mapy. Server je konkurentní, tzn., že na něm může běžet více úloh současně, avšak počet je omezen, aby nedošlo k zahlcení. Každá úloha se dokončí bez nutné interakce s klientem. Klient se může dotazovat na informace o postupu úloh, aby mohl uživateli dát vědět mj. kolik map je zpracovaných z celkového počtu.

6.1 Úloha

Každá úloha se skládá z množiny wafer map a množiny algoritmů, které se nad ní mají vykonat. Úloha běží na serveru jako samostatné vlákno.
Množina wafer map je identifikována pomocí následující selekce:

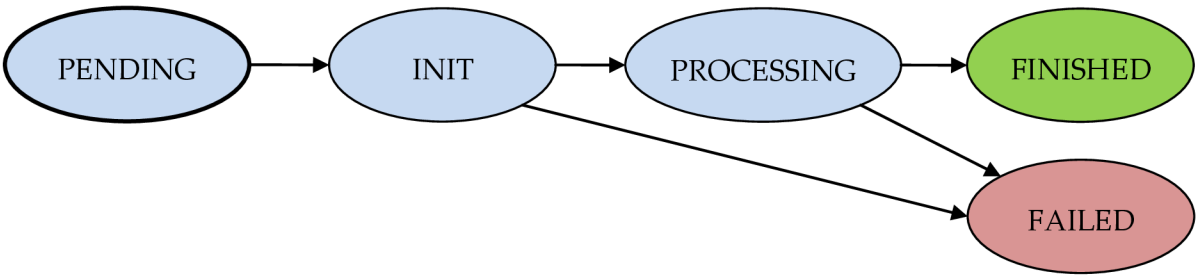
- **Lot ID** – jednoznačný identifikátor sady
- **Wafer ID** – číslo desky, jednoznačné v rámci sady
- **Probe end date** – datum a čas posledního zpracování

Povinně se musí zadat pouze *lot ID*, tím vybereme všechny wafer mapy v sadě. Pokud zadáme *wafer ID*, zúžíme výběr na jedinou mapu. V úložišti Map Vault jsou ukládány wafer mapy s celou jejich historií, což nám dovoluje zadat další kritérium, kterým je *probe end date*. Vyplnit ho můžeme pouze, pokud je zadané *lot ID* a *wafer ID*. Necháme-li *probe end date* prázdné, vybere se automaticky nejnovější mapa.

Množinu algoritmů budeme nazývat filtrem a zadané algoritmy se vykonávají v pořadí, které určí klient. V případě více algoritmů ve filtru se vždy výsledek zpracování předchozího algoritmu předá ke zpracování následujícímu algoritmu.

6.2 Stav úlohy

Každá úloha se nachází ve stavech znázorněných na obrázku 11. Počátečním stavem je *PENDING*, konečnými stavy jsou *FINISHED* nebo *FAILED*, podle toho, zda byla úloha dokončena bez chyb či nikoli.



Obrázek 11 - Diagram stavů úlohy

- *PENDING* – úloha čeká ve frontě na zpracování
- *INIT* – probíhá inicializační fáze
- *PROCESSING* – probíhá zpracování map
- *FINISHED* – úloha je úspěšně dokončena
- *FAILED* – úlohu se nepodařilo dokončit

6.3 Konfigurační soubor

Pro konfiguraci služby je určen soubor zvaný `config.xml`, který je v rámci projektu umístěn v `src/main/resources`. Při nasazení na aplikační server se rozbalí do kořenového adresáře se službou.

Formát

```
<?xml version="1.0" encoding="UTF-8"?>
<PostProcessingWSConfig>
 <MapVaultWSLocation>http://cesta.k.map/vault</MapVaultWSLocation>
 <MaxProcessingThreads>16</MaxProcessingThreads>
 <TaskExpireTimeout>1000</TaskExpireTimeout><!-- seconds -->
 <WatchdogTimeout>10</WatchdogTimeout><!-- seconds -->
</PostProcessingWSConfig>
```

Význam konfiguračních elementů

- `MapVaultWSLocation` – cesta k webové službě Map Vault
- `MaxProcessingThreads` – maximální počet současně běžících úloh
- `TaskExpireTimeout` – minimální čas v sekundách, po který je stav úlohy po jejím dokončení udržován na serveru
- `WatchdogTimeout` – maximální čas v sekundách, po který může běžet algoritmus

Výchozí hodnoty

- `MapVaultWSLocation` – *povinná položka*
- `MaxProcessingThreads` = 16
- `TaskExpireTimeout` = 1000
- `WatchdogTimeout` = 10

Pokud je konfigurační soubor chybný, je do logu zaznamenána chyba. Služba pak neběží standardním způsobem a každá zavolaná WS metoda vyvolá výjimku *RuntimeException*.

6.4 Log

Důležité události služba zaznamenává do logu. Textový soubor s logem je situován do standardního umístění pro logy, které určí aplikační server. V případě Tomcat je to v `%TOMCAT_HOME%/bin/Log/PostProcessingWS.log`. Pro logování je použita knihovna *log4j*.

6.5 Systém vláken

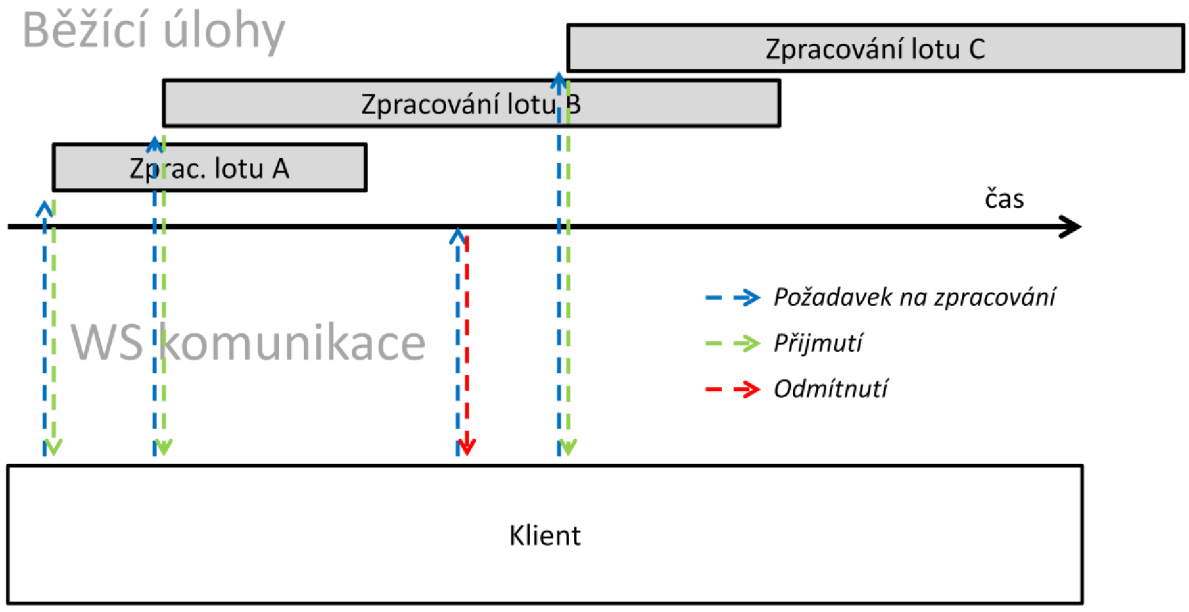
JAX-WS je abstrakcí nad servlety a dokáže zpracovávat požadavky konkurentně. Funguje to tak, že každý požadavek servlet obslouží a vrátí klientovi odpověď. Samotné vykonávání úlohy (zpracování map) je však časově náročný proces a je nepřípustné, aby klient musel čekat na dokončení celého zpracování. Proto jsem přistoupil k modelu, kde jsou odděleny požadavky od zpracování. Klient zadá požadavek na zpracování úlohy a dostane ihned potvrzení, zda je požadavek přijat. Na serveru se potom požadovaná úloha rozběhne na pozadí. Systém požadavků a zpracovávání je znázorněn na obrázku 12.

Implementace tohoto modelu v Java EE přináší jednu zajímavou komplikaci. Prvotní, ale nesprávná, myšlenka pro realizaci spočívá ve vytvoření vlákna s úlohou, které necháme rozběhnout v době požadavku. Toto vlákno by pak běželo i po uzavření spojení s klientem. Problémem je, že v Java EE se zakazuje manuální správa vláken, což dokazuje následující restrikce:

> *„The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread, or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.“* [15]

Řešení jsem nalezl v systému *thread pool*, kterým Java disponuje. Umožňuje správu vláken na vyšší úrovni abstrakce. Do *thread pool* pak zadáváme požadavky na zpracování úloh a JVM si pro ně sám vytváří a spravuje vlákna. Tím se distancujeme od manuální správy vláken a vyhneme se tomu, že se přeruší všechna vlákna vytvořená během vykonávání webové metody.

Konkrétně *fixed thread pool*, který jsem použil, umožňuje zadat maximální počet současně běžících vláken. Tato vlákna si vytvoří jen při inicializaci a pak je používá znovu, což navíc přináší výhodu nižší režie při vytváření vláken.



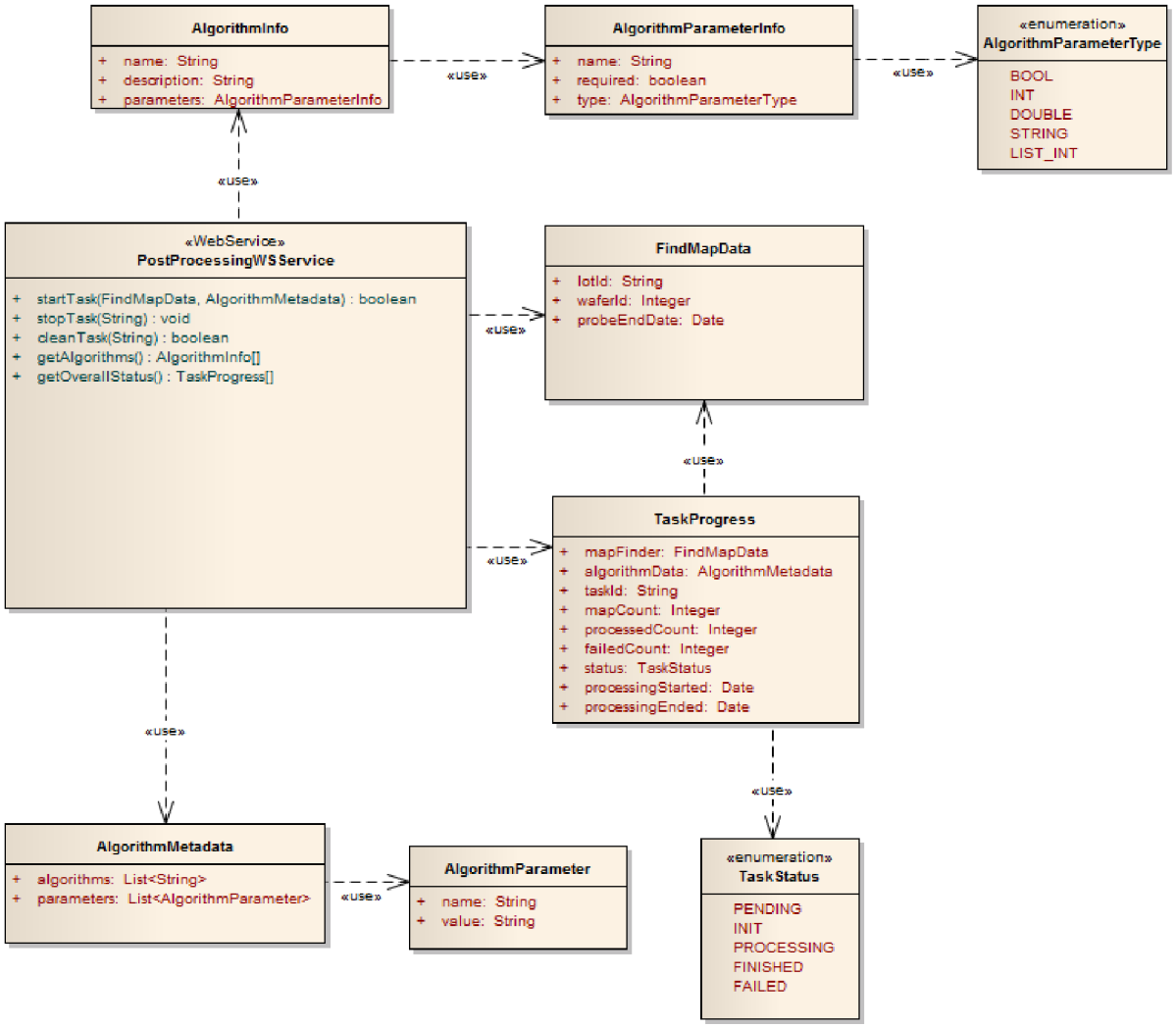
Obrázek 12 - Znázornění požadavků a úloh

6.6 Rozhraní webové služby

Přes metody webové služby je možné serverovou aplikaci kompletně ovládat. Celé rozhraní je navrženo univerzálně, obsahuje jen několik metod a datových typů. Znalost rozhraní je důležitá pro implementaci klienta.

6.6.1 Diagram tříd

Na obrázku 13 vidíme UML diagram tříd pro rozhraní webové služby. Všechny prvky tohoto modelu jsou obsaženy v popisu rozhraní webové služby (formát *WSDL*).



Obrázek 13 - Diagram tříd pro rozhraní webové služby

Koncovým bodem (endpoint) pro webovou službu je třída `PostProcessingWSService`. Právě ta obsahuje webové metody na obsluhu serveru.

Další třídy v diagramu, které třída `PostProcessingWSService` používá, jsou pouze nosičem dat pro předání webovým metodám.

6.6.2 WS metody

startTask

Zadá požadavek na zpracování úlohy. Množina map se identifikuje pomocí `FindMapData`, algoritmus a parametry pro něj jsou předány v `AlgorithmMetadata`. Pokud metoda vrátí `true`, požadavek byl přijat a zařazen do fronty na zpracování, v opačném případě byl požadavek na zpracování odmítnut. Důvodem odmítnutí může být kolize s jinou zpracovávanou úlohou nebo nesprávný formát parametrů, které vyžadují algoritmy.

stopTask

Zastaví čekající nebo již běžící úlohu, danou pomocí identifikátoru úlohy. Úloha tím přejde do stavu *FAILED*.

cleanTask

Smaže dokončenou úlohu ze serveru, danou pomocí identifikátoru úlohy. Čekající nebo běžící úlohy nelze smazat, musíme je nejdříve zastavit pomocí metody `stopTask`.

getOverallStatus

Zjistí informace o postupu všech úloh, které jsou vráceny v seznamu objektů typu `TaskProgress`.

getAlgorithms

Získá informace o všech aplikovatelných algoritmech, které vrátí v seznamu objektů typu `AlgorithmInfo`.

6.6.3 WS třídy

Třídy jsou ve WS použity pouze k strukturovanému přenosu informací. K nadepsanému názvu třídy jsou v odrážkách uvedeny jejich vlastnosti.

FindMapData

- `String` **`lotId`** – identifikátor sady
- `Integer` **`waferId`** – identifikátor waferu
- `Date` **`probeEndDate`** – čas zpracování mapy

Obsahuje informace pro vyhledání mapy. Nastavením příslušné vlastnosti na `null` se tato vlastnost nepoužije. Například pokud se zadá jen identifikátor sady, vyhledají se všechny mapy v sadě, pokud se zadají všechny vlastnosti, je vyhledána zpravidla pouze jedna mapa.

AlgorithmMetadata

- `List<String>` **`algorithms`** – seznam algoritmů, které se mají vykonat, pořadí vykonávání je stejné jako pořadí v seznamu.
- `List<AlgorithmParameter>` **`parameters`** – parametry předávané algoritmům

Používá se pro specifikaci filtru algoritmů, které se mají vykonat. V parametrech jsou uložena dodatečná řídicí data. Parametry jsou sdílené pro celý filtr algoritmů, tzn., že všem algoritmům ve filtru jsou předána stejná data. Specifické parametry si jednotlivé algoritmy musí rozlišit pomocí prefixu. Naopak globální parametry mohou užívat všechny algoritmy ve filtru.

AlgorithmParameter

- `String` **`name`** – název parametru
- `String` **`value`** – hodnota parametru

Hodnoty všech typů parametrů jsou zakódovány do řetězce. Pokud je vyžadován specifičtější datový typ, jako `INT` nebo `DOUBLE`, pak se očekává hodnota s určitým formátem. Více informací bude uvedeno v popisu výčtu `AlgorithmParameterType`.

AlgorithmInfo

- `String` **`name`** – unikátní název algoritmu
- `String` **`label`** – uživatelský název
- `String` **`description`** – stručný popis
- `AlgorithmParameterInfo[]` **`parameters`** – informace o parametrech

Nese nezbytné informace o algoritmu pro podporu klientského uživatelského rozhraní.

AlgorithmParameterInfo

- `String` **`name`** – název parametru
- `boolean` **`required`** – příznak, zda je povinný
- `AlgorithmParameterType` **`type`** – datový typ

Slouží pro zobrazení a validaci parametrů, které se předávají modulům s algoritmy. V uživatelském rozhraní se pak mohou zobrazit všechny povinné i nepovinné parametry, které může uživatel vyplnit. Validace později předaných parametrů je prováděna na serveru. Pokud není splněna podmínka povinnosti nebo datového typu, je požadavek na zpracování úlohy odmítnut.

TaskProgress

- `FindMapData` **`mapFinder`** – data pro vyhledání mapy, která byla použita pro tuto úlohu
- `AlgorithmMetadata` **`algorithmData`** – filtr algoritmů použitý pro tuto úlohu
- `String` **`taskId`** – identifikátor úlohy
- `Integer` **`mapCount`** – počet map pro tuto úlohu
- `Integer` **`processedCount`** – počet úspěšně zpracovaných map
- `Integer` **`failedCount`** – počet map, které se nepodařilo zpracovat
- `TaskStatus` **`status`** – aktuální stav úlohy
- `Date` **`processingStarted`** – čas, kdy bylo započato zpracování
- `Date` **`processingEnded`** – čas, kdy bylo dokončeno zpracování

6.6.4 WS výčty

AlgorithmParameterType

- **`STRING`** – parametr nemá žádná omezení
- **`BOOL`** – je vyžadována řetězcová hodnota `true` nebo `false`
- **`INT`** – je vyžadována celočíselná hodnota
- **`DOUBLE`** – je vyžadováno reálné číslo s desetinnou tečkou
- **`LIST_INT`** – je vyžadován seznam 0 nebo více celočíselných hodnot, které jsou odděleny středníkem (`;`)

Vyjadřuje požadovaný typ parametru, který je předán modulu s algoritmem. Všechny parametry se předávají univerzálně řetězcovou hodnotou a dekódování probíhá až na serveru. Tyto typy určují očekávaný formát parametru. Při nedodržení formátu je požadavek na zpracování úlohy odmítnut.

TaskStatus

- **`PENDING`** – úloha čeká ve frontě na zpracování
- **`INIT`** – probíhá inicializační fáze
- **`PROCESSING`** – probíhá zpracování map
- **`FINISHED`** – úloha je úspěšně dokončena
- **`FAILED`** – úlohu se nepodařilo dokončit

Vyjadřuje aktuální stav úlohy. Vysvětlení stavů úlohy je v kapitole 6.2.

6.7 Systém modularity algoritmů

Jednou z předností systému má být jeho pružnost vůči technologickým změnám. Modularita procesních algoritmů je hlavní prostředek, který to má umožnit. Moduly s algoritmy lze jednoduše implementovat bez hluboké znalosti systému. Zároveň modul funguje jako uzavřený box a serverová aplikace je do jisté míry imunní proti potížím vzniklým za běhu modulu.

6.7.1 Vytvoření modulu s algoritmem

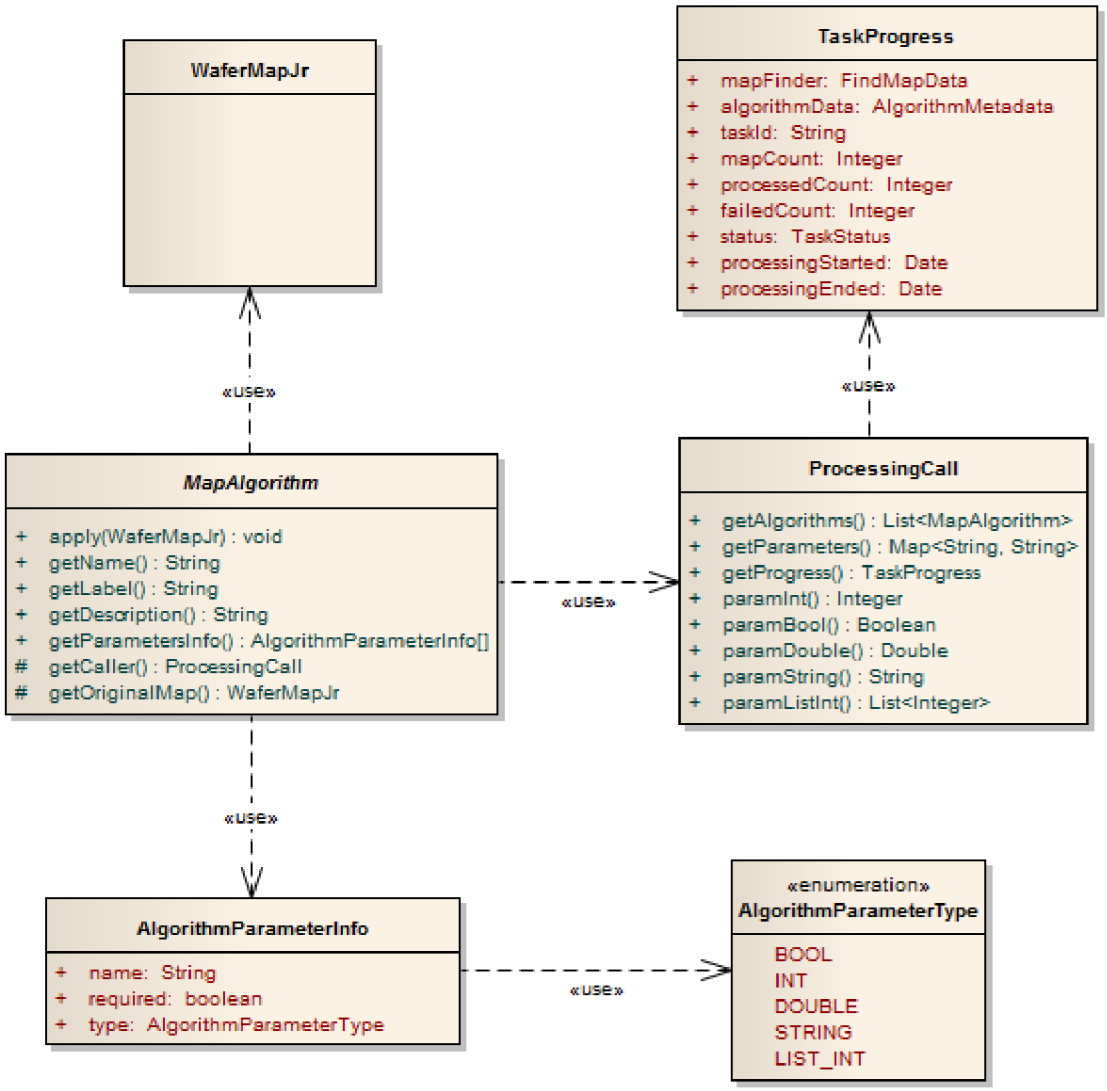
Pro implementaci nového algoritmu jsou nutné pouze tyto kroky:

1) Vytvořit novou třídu ve jmenném prostoru `com.onsemi.cim.apps.postprocessingws.mapalgorithms`.
2) Touto třídou rozšířit abstraktní třídu `MapAlgorithm`.
3) Implementovat tělo algoritmu do metody `apply(WaferMapJr waferMap)`.

Po provedení tohoto postupu se algoritmus automaticky objeví v systému, protože třídy s implementovanými algoritmy jsou automaticky skenovány. Funguje to tak, že se postupně projdou všechny třídy ve sledovaném jmenném prostoru a z nich se vyberou ty třídy, které jsou zděděny z `MapAlgorithm`.

6.7.2 Diagram tříd

Na obrázku 14 jsou znázorněny podstatné třídy, které jsou v interakci s modulem



Obrázek 14 - Diagram tříd, které jsou v interakci s modulem

Každý modul s algoritmem musí dědit z abstraktní třídy `MapAlgorithm`. Pomocí této třídy lze přistupovat k detailům volání (`ProcessingCall`), které obsahuje předávané parametry a informaci o postupu (`TaskProgress`). Předávané parametry jsou validovány podle informací v objektech typu `AlgorithmParameterInfo`. Wafer mapa je zde reprezentována třídou `WaferMapJr`.

6.7.3 Třída MapAlgorithm

Jelikož je `MapAlgorithm` základní třídou pro vytváření vlastních algoritmů pro zpracování wafer map, popíšu její metody, které lze implementovat nebo použít.

apply

```
public abstract void apply(WaferMapJr wfrMap);
```

Jedná se o jedinou abstraktní metodu, kterou musí dědic povinně implementovat. V této metodě je umístěn kód, který algoritmus vykoná nad jednou wafer mapou. Wafer mapa je předána jako parametr. Tuto mapu algoritmus zmodifikuje a předaný objekt se tedy zároveň použije jako výstup. Třída `WaferMapJr` je univerzálně používaná ve všech projektech, kde se pracuje s wafer mapou.

getName

```
public final String getName();
```

Vrátí jednoznačný název algoritmu, který je dán názvem třídy s algoritmem.

getLabel

```
public String getLabel();
```

Toto je uživatelský název algoritmu, který se pak může použít v GUI u klienta. Standardně vrací to samé, jako `getName()`, ale tuto metodu lze přepsat.

getDescription

```
public String getDescription();
```

Měla by vrátit krátký popis algoritmu, ve kterém se můžou vyskytovat i základní HTML značky.

getOriginalMap

```
protected final WaferMapJr getOriginalMap();
```

Vrací originální načtenou mapu, do které nejsou zahrnuty změny, které byly doposud algoritmem provedeny. To je dobré, pokud si algoritmus mapu mění, ale zároveň potřebuje

vycházet z původních dat. Je to hluboká kopie wafer mapy, kterou systém provede před zavoláním metody `apply`.

getCaller

```
protected final ProcessingCall getCaller();
```

Vrací objekt s parametry volání. Obsahuje filtr algoritmů, aktuální status úlohy a parametry.

getParametersInfo

```
public AlgorithmParameterInfo[] getParametersInfo();
```

Zde se definuje seznam algoritmů, které parametr očekává. Pomocí třídy AlgorithmParameterInfo se nastaví název parametru, jeho typ a příznak povinnosti. Při zadání požadavku pak systém uživatelem zadané parametry automaticky zkontroluje podle těchto specifik.

6.7.4 Třída ProcessingCall

Tato třída si zaslouží pozornost zejména proto, že přes ni lze přistupovat k parametrům, které se předávají algoritmům. Popíšu veřejné metody, které jsou užitečné pro použití uvnitř implementace algoritmu.

paramString

```
public String paramString(String name);
```

Vrací řetězcový parametr adresovaný názvem v `name`. V případě, že není definován, vrací `null`.

paramBool

```
public boolean paramBool(String name);
```

Vrací dekódovaný parametr, který má být typu `BOOL`. V případě, že není definován nebo nelze dekódovat, vrací `null`.

paramInt

```
public Integer paramInt(String name);
```

Vrací dekódovaný parametr, který má být typu `INT`. V případě, že není definován nebo nelze dekódovat, vrací `null`.

paramDouble

```
public Double paramDouble(String name);
```

Vrací dekódovaný parametr, který má být typu `DOUBLE`. V případě, že není definován nebo nelze dekódovat, vrací `null`.

getParameters

```
public Map<String, String> getParameters();
```

Vrací mapu všech parametrů, kde klíčem je název parametru a hodnotou je originální řetězcová reprezentace.

6.7.5 Třída WaferMapJr

Tato třída je univerzálně používaná pro wafer mapu. Umožňuje získat mapu binů, informace o geometrie waferu a další informace týkající se naměřeného waferu.

Je součástí projektu se stejnojmenným názvem *WaferMapJr*, který je autorským dílem společnosti ON Semiconductor. Proto popis jejího rozhraní není součástí této práce.

6.8 Průběh zpracování úlohy

Po přijetí požadavku na novou úlohu serverem běží zpracování nezávisle až do dokončení bez nutnosti interakce klienta. Klient se v průběhu zpracování může dotazovat na postup úlohy, popřípadě ji ukončit.

6.8.1 Informace o postupu

Informace o postupu je reprezentována objektem typu `ProcessProgress` přiřazený k dané úloze. Kromě stavu úlohy jsou tu další informace, např. kolik map je již zpracovaných z celkového počtu a také stavové zprávy, které mohou být využity pro přenos podrobností o chybách.

6.8.2 Výjimky v modulu s algoritmem

Pokud dojde uvnitř modulu s algoritmem při zpracovávání wafer mapy k výjimce, provedou se následující akce:

1) Zpracování mapy je přerušeno, aktuální mapa se již neuloží zpět do map vaultu.
2) Text výjimky, společně s identifikací algoritmu a aktuálně zpracovávané mapy, je uložen do logu a do stavových zpráv.
3) Pokračuje se zpracováním následující mapy v úloze a další chod již není ovlivněn.
4) Cílovým stavem úlohy bude *FAILED*.

Tím si systém zajišťuje odolnost vůči chybám, které mohou v modulu nastat. Je pak na programátorovi modulů, aby problémové moduly opravil.

6.8.3 Watchdog

Watchdog, neboli hlídací časovač, je dalším mechanismem pro zvýšení odolnosti vůči chybám v modulech. Uvnitř modulu s algoritmem se může stát téměř cokoli včetně zacyklení v nekonečné smyčce. Aby byl server imunní proti zbytečnému přetěžování, je deklarována maximální doba, po kterou může algoritmus běžet. Jakmile je překročena, algoritmus je okamžitě ukončen. Zpracovávaná mapa se samozřejmě již neuloží a chyba je zaznamenána do logu i do stavové zprávy, stejně jako v případě výjimky. Je pokračováno zpracováním následující mapy v úloze a po dokončení úlohy bude cílovým stavem *FAILED*.

7 Implementace klientské aplikace

Klientská aplikace má sloužit pro manuální obsluhu serverové aplikace. Je navržena jako webová stránka, tak aby byla spustitelná v běžných webových prohlížečích. Účelem je manuální správa serverové aplikace. Jsou použity tyto technologie:

- JSP
- XHTML
- CSS
- JavaScript
- JSON

Jazykem pro uživatelské rozhraní je angličtina, protože program se plánuje užívat ve všech pobočkách ON Semiconductor.

7.1 Konfigurační soubor

Pro konfiguraci slouží soubor `config.xml`. Stejně jako u serverové aplikace je v rámci projektu umístěn v `src/main/resources`.

Formát

```
<?xml version="1.0" encoding="UTF-8"?>
<PostProcessingControllerConfig>
 <PostProcessingWSLocation>http://cesta.k.post.processing/ws
</PostProcessingWSLocation>
 <StatusRefreshInterval>500</StatusRefreshInterval><!-- milliseconds -->
</PostProcessingControllerConfig>
```

Význam konfiguračních elementů

- `PostProcessingWSLocation` – umístění webové služby *Post Processing WS*.
- `StatusRefreshInterval` – interval pro obnovení statusu v milisekundách

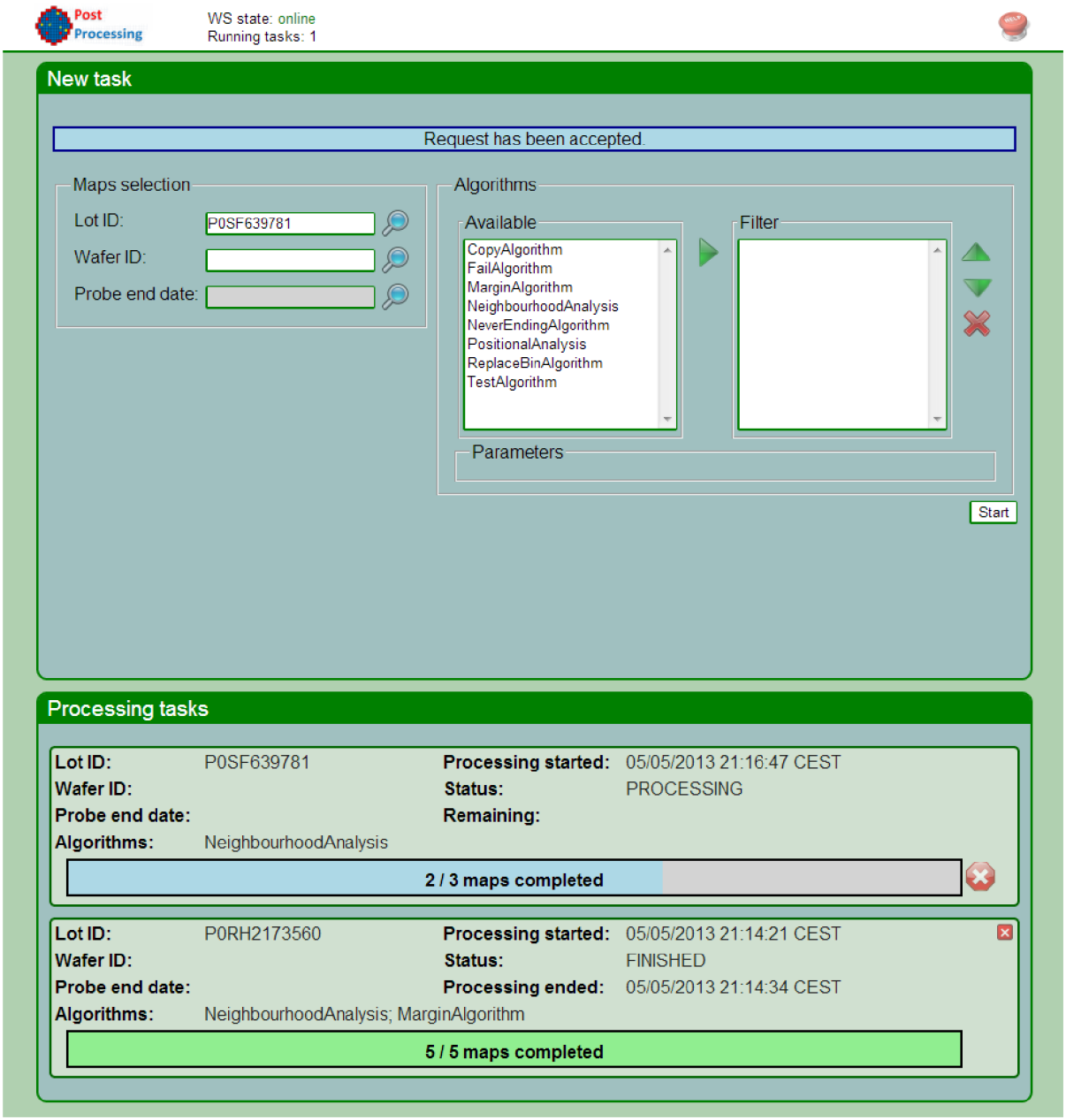
Výchozí hodnoty

- `PostProcessingWSLocation` – *povinná položka*
- `StatusRefreshInterval` = 500

7.2 Ovládání aplikace

Při navrhování byl kladen důraz na maximální jednoduchost. Celá funkčnost je zobrazena na jediné stránce, kterou můžete vidět na obrázku 15.

Nahoře je stavový řádek, ve kterém máme přehled o dostupnosti webové služby a o počtu běžících úloh.



Obrázek 15 - Vzhled klientské webové aplikace

Webová aplikace má dva hlavní účely, a to zadávání nových úloh a sledování probíhajících úloh. Na oba účely jsou vyhrazeny samostatné boxy.

7.2.1 Spuštění nové úlohy

Údaje pro spuštění nové úlohy zadáme do boxu *New task*. V levé části *Maps selection* vybereme mapy, které chceme zpracovat. Identifikujeme je pomocí *lot ID* (povinné) a popřípadě můžeme rozšířit kritéria o *wafer ID* a *probe end date*. Způsob selekce map již byl zmíněn v kapitole 6.1.

Algoritmy, které se mají vykonat nad množinou vybraných map, přesouváme z dostupných algoritmů (*Available*) do filtru (*Filter*). Učinit můžeme jednak pomocí tlačítka šipky doprava, ale i pomocí klávesy mezerník. U algoritmů ve filtru záleží na pořadí, výsledek zpracování každého algoritmu se předá následujícímu algoritmu. Ve filtru můžeme toto pořadí měnit nebo jednotlivé položky mazat pomocí tlačítek vpravo (mazání i pomocí klávesy *delete*).

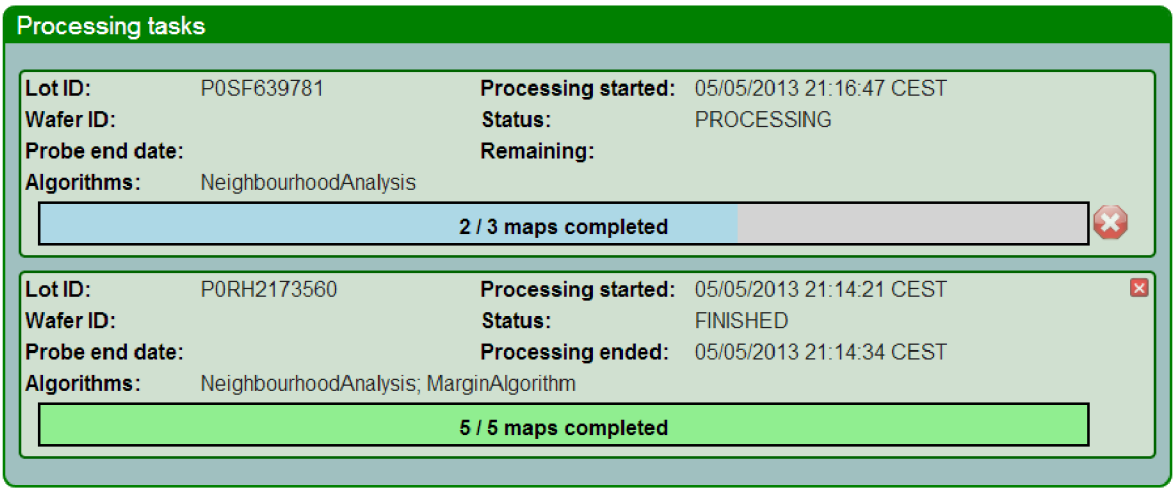
Parametry, které algoritmy požadují, se zobrazují v sekci *Parameters*. Dynamicky se mění podle toho, které algoritmy jsou ve filtru zrovna vybrány. Parametry jsou sdílené, tzn., že každý algoritmus může mít přístup ke všem předaným parametrům. Ve většině případů však parametr používá pouze jediný algoritmus. Je zavedena koncepce, že pokud parametr není univerzální, jeho název má prefix odpovídající algoritmu, ve kterém je použit.

7.2.2 Sledování úloh

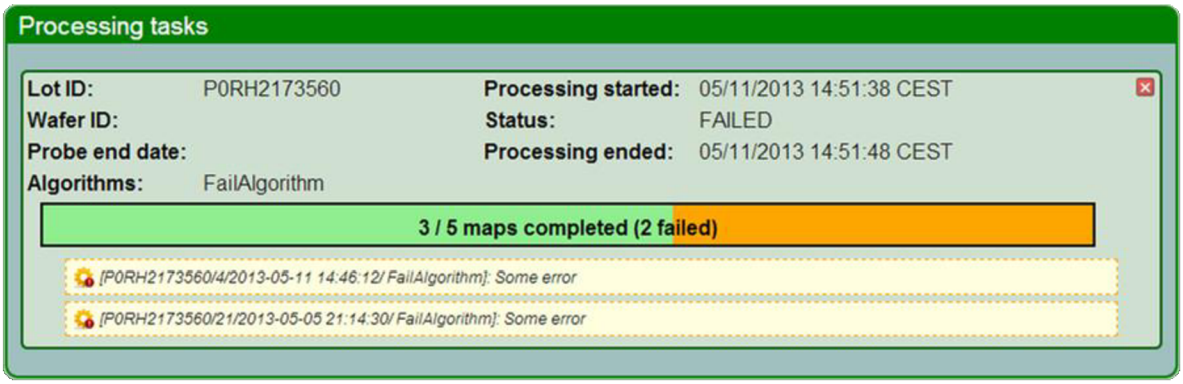
V boxu *Processing tasks* (obrázek 16) vidíme úlohy, které jsou na serveru. To zahrnuje čekající úlohy, běžící úlohy i dokončené úlohy. Ty, které právě běží mají zobrazený indikátor průběhu (progress bar).

Nechybí ani možnost úlohu zastavit, a to pomocí červeného tlačítka napravo od indikátoru průběhu. Úloha, která je dokončena, je na serveru udržována ještě po určitou dobu (výchozí hodnota je 1000 s), aby uživatel mohl zpozorovat informace o zpracování. Odstranit se dá však i dříve přes tlačítko s křížkem v horní části úlohy.

Pokud při zpracování úlohy dojde k určitým chybám, uživatel se to dozví pomocí status zpráv zobrazených pod indikátorem průběhu. V indikátoru průběhu je kromě počtu dokončených map informace i o počtu map, které se nepodařilo dokončit (obrázek 17).



Obrázek 16 - Přehled úloh na serveru



Obrázek 17 - Ukázka úlohy, která byla dokončena s chybami

7.3 Implementační detaily

Pro správnou funkci webu je nutné mít aktivovaný JavaScript, kterým je na stránce vykreslována většina obsahu. Web je kompatibilní s následujícími prohlížeči, na kterých byl testován:

- Google Chrome
- Mozilla Firefox
- Opera
- Microsoft Internet Explorer, verze 9 nebo vyšší

Největší problém byl zprovoznit web pro IE, který se vychyluje standardům pro JavaScript a CSS. Částečně je web funkční i pod IE verze 8 (za pomoci obcházení nekompletního DOM modelu), ale doporučuji používat verzi 9.

Celé rozhraní je zobrazeno na jedné webové stránce. Jak již bylo vidět na obrázku 15, stránka je rozdělena na 3 logické celky:

- Vrchní lišta s přehledem
- Zadávání nové úlohy – box *New task*
- Seznam úloh na serveru – box *Processing tasks*

Každý z těchto celků je implementován specifickým způsobem.

7.3.1 Vrchní lišta s přehledem

Vrchní lišta má na obrazovce fixní pozici, tzn., že je vidět pořád na stejném místě i při posouvání stránky. Stav serveru je kontrolován pomocí AJAX, kód je umístěn v `js/serverstatus.js`. Dotazování probíhá pravidelně po době stanovené konfigurační konstantou `StatusRefreshInterval`. Informace jsou přenášeny ve formátu JSON, generované JSP skriptem `serverstatus.jsp`. Ten vrátí aktuální údaje o serveru a úlohách, které jsou použity i v boxu *Processing tasks*.

7.3.2 Box *New task*

Tato část umožňující zadávání nové úlohy je umístěna v samostatném rámci. Je to učiněno z toho důvodu, aby byl uživatel ušetřen od zbytečného překreslování celé stránky. V celém formuláři je nejzajímavější sekcí *Algorithms*. Vybírání algoritmů do filtru je realizováno dvěma elementy *select*. Pokud tlačítkem přesuneme algoritmus do filtru, pomocí JavaScriptu se zkopíruje položka z pravého elementu *select* do levého. Aktualizuje se také seznam parametrů (sekce *Parameters*), kde se vygenerují textová pole pro všechny parametry, které algoritmy ve filtru vyžadují.

7.3.3 Box *Processing tasks*

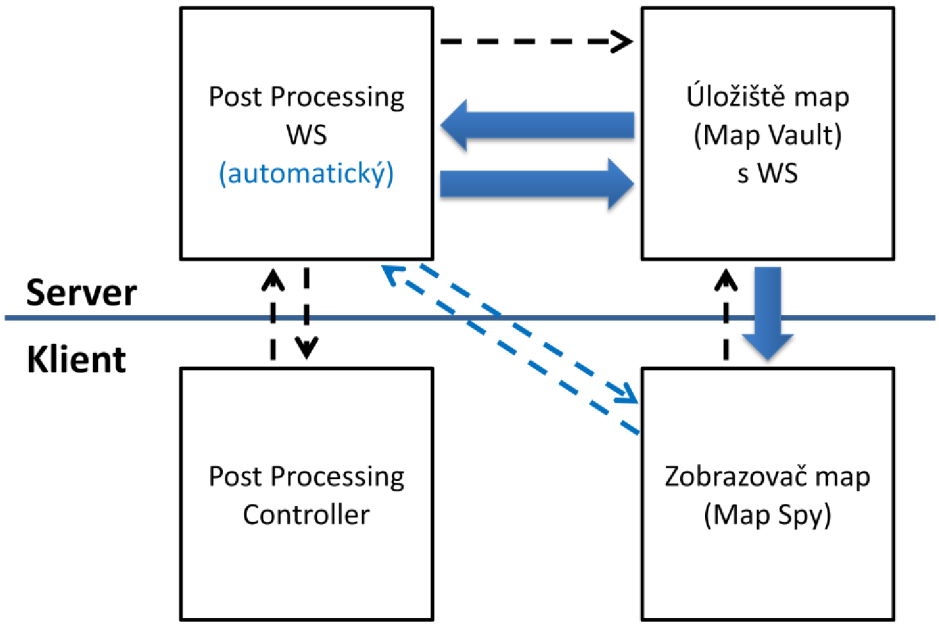
Uživatel vidí poměrně plynule aktuální průběh zpracování úloh díky využití AJAX. Celý box s postupem úlohy je generován JavaScript kódem v `serverstatus.js` Informace o běžících úlohách jsou získávány ze `serverstatus.jsp`. Tlačítka pro zastavení, resp. odstranění úlohy odkazují na skripty `stoptask.jsp`, resp. `cleantask.jsp`. Aby se nemusela aktualizovat celá stránka, je cílem odkazu neviditelný rámec.

8 Závěr

Softwarový systém *Post Processing* byl navrhnut a implementován podle všech zadaných požadavků. Serverová aplikace má podle zadání zajištěné WS rozhraní pro jednotný a kontrolovaný přístup k úlohám, přístup k odpovídajícím datovým úložištím, mechanizmus asynchronního zpracování i modularitu pro algoritmy, které jsou určeny na zpracování wafer map. Klientská aplikace je podle zadání jednoduchým webovým rozhraním pro plnohodnotnou obsluhu serveru. Navíc byly implementovány některé z algoritmů, konkrétně *Neighbourhood Analysis* a *Positional Analysis*.

Vytvořený systém byl předán společnosti ON Semiconductor, která ho plánuje rozšířit a řádně otestovat. Poté bude nasazen do reálného produkčního provozu. Plánovaná rozšíření se týkají podpory jiných formátů wafer map (zatím systém podporuje pouze WMXML) a implementaci dalších algoritmů, např. DPAT.

Aplikace *Post Processing WS* bude v budoucnu spolupracovat s aplikací *Map Spy*, do které bude zabudována podpora pro zpracování dat. Vizí je také plná automatizace aplikace *Post Processing WS*, která by průběžně detekovala nově zaznamenané wafer mapy a vyhodnocovala automaticky, které algoritmy se na nich mají aplikovat. Díky aplikaci *Post Processing Controller* se bude server obsluhovat pouze manuálně. Budoucí vývoj shrnuje blokové schéma na obrázku 18.



Obrázek 18 - Budoucí vývoj v blokovém schématu

Slovníček pojmů

Algoritmus	*modul s algoritmem pro zpracování map, pokud není uvedeno jinak*
Bin	*číslo vyjadřující typ chyby na čipu*
Fazeta	*část, kde je kruhový půdorys waferu seřezán do roviny pro orientaci*
Informace o postupu	*informace o stavu, průběhu a dalších detailech úlohy*
JVM	*Java Virtual Machine – virtuální stroj pro vykonávání Java kódu*
Lot	*sada waferů (výrobní várka)*
Monokrystal	*válcovitý kus čistého křemíku, který má krystalickou strukturu v celém svém objemu*
Multisite	*obdélníková oblast čipů měřených najednou*
Notch	*trojúhelníkový zářez do kruhového půdorysu waferu pro orientaci*
Probe end date	*datum a čas posledního zpracování wafer mapy*
Retikl	*fotomaska, která se aplikuje na wafer*
Stavová zpráva	*chybové zprávy, které jsou součástí informace o postupu*
Tester	*zařízení, které elektricky testuje čipy na waferu*
Úloha	*množina wafer map, která se má zpracovat množinou algoritmů*
Výtěžnost (yield)	*poměr počtu použitelných čipů vůči počtu všem potenciálně použitelných čipů*
Wafer / deska	*polovodičová deska*
Wafer mapa	*mapa výsledků měření na waferu*

Literatura

[1] IT centrum sdílených služeb. *ON Semiconductor* [online]. [cit. 15. 4. 2013]. Dostupné z http://www.onsemi.com/PowerSolutions/content.do?id=16797

[2] Mirza, A.: *Spatial Yield Modeling for Semiconductor Wafers*. Massechusetts Institute od Technology, 1995

[3] Web service. *Wikipedia* [online]. [cit. 19. 4. 2013]. Dostupné z http://en.wikipedia.org/wiki/Web_service

[4] Web Services Architecture. *W3C Working Group Note* [online]. W3C, ©2004. [cit. 5. 4. 2013]. Dostupné z http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[5] SOAP. *Wikipedia* [online]. [cit. 9. 5. 2013]. Dostupné z http://en.wikipedia.org/wiki/SOAP

[6] TIOBE Programming Community Index for May 2013. *TIOBE Software BV* [online]. [cit. 10. 5. 2013]. Dostupné z http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[7] Herout, Pavel. *Učebnice jazyka Java*. České Budějovice: Kopp, 2011. ISBN 978-8072323982.

[8] Java API for XML Web Services. *Wikipedia* [online]. [cit. 19. 4. 2013]. Dostupné z http://en.wikipedia.org/wiki/Java_API_for_XML_Web_Services

[9] JavaServer Pages. *Wikipedia* [online]. [cit. 19. 4. 2013]. Dostupné z http://en.wikipedia.org/wiki/JavaServer_Pages

[10] JavaScript. *Wikipedia* [online]. [cit. 10. 5. 2013]. Dostupné z http://en.wikipedia.org/wiki/JavaScript

[11] AJAX. *Wikipedia* [online]. [cit. 10. 5. 2013]. Dostupné z http://en.wikipedia.org/wiki/Ajax_(programming)

[12] Apache Tomcat. *Wikipedia* [online]. [cit. 2. 5. 2013]. Dostupné z http://en.wikipedia.org/wiki/Apache_Tomcat

[13] Apache Maven. *Wikipedia* [online]. [cit. 12. 5. 2013]. Dostupné z http://en.wikipedia.org/wiki/Apache_Maven

[14] Mercurial. *Wikipedia* [online]. [cit. 10. 5. 2013]. Dostupné z http://en.wikipedia.org/wiki/Mercurial

[15] NetBeans. *Wikipedia* [online]. [cit. 12. 5. 2013]. Dostupné z http://en.wikipedia.org/wiki/Netbeans

[16] Mison, D.; Rooskov, I. a Wulf, J.: JBoss Enterprise Application Platform 4.3. *Red Hat* [online]. Red Hat 2008 [cit. 8. 5. 2013]. Dostupné z https://access.redhat.com/site/documentation/en-US/JBoss_Enterprise_Application_Platform/4.3/html/Common_Criteria_Configuration_Guide/Common_Criteria_Guide-Developer_Guidelines-general_restrictions.html

[17] Eckel, Bruce. *Thinking in Java (3rd Edition) Revision 4.0* [online]. Prentice Hall 2002 [cit. 15. 3. 2013]. ISBN 978-0131002876. Dostupné z http://www.mindviewinc.com/Books/downloads.html

[18] Amstrong, Eric. *The Java Web Services Tutorial*. Santa Clara: Pearson Education, 2005. ISBN 978-0201768114.

[19] Web Services Description Language. *Wikipedia* [online]. [cit. 12. 5. 2013]. Dostupné z http://en.wikipedia.org/wiki/Web_Services_Description_Language

Seznam příloh

Příloha 1: Pseudokódy

A. Neighbourhood Analysis

```
// Hlavní funkce, která je volána
void apply(WaferMap waferMap)
{
      // Vytvoří hlubokou kopii wafer mapy
      WaferMap originalMap = waferMap.copy();

      for (int r = 0; r < ROWS; r++)
      {
            for (int c = 0; c < COLLUMNS; c++)
            {
                  int bin = originalMap.getBin(r, c);

                  if (isBadBin(bin))
                  {
                        // Pokud najde vadný čip, označí všechny v okolí
                        surroundBadChip(waferMap, r, c);
                  }
            }
      }
}

// Označí všechny čipy v osmiokolí od dané pozice
void surroundBadChip(WaferMap waferMap, int row, int col)
{
      markBadChip(waferMap, row - 1, col - 1);
      markBadChip(waferMap, row - 1, col);
      markBadChip(waferMap, row - 1, col + 1);
      markBadChip(waferMap, row, col - 1);
      markBadChip(waferMap, row, col + 1);
      markBadChip(waferMap, row + 1, col - 1);
      markBadChip(waferMap, row + 1, col);
      markBadChip(waferMap, row + 1, col + 1);
}
```

```
void markBadChip(WaferMap waferMap, int row, int collumn)
{
      waferMap.setBin(MARK_BIN, row, collumn);
}
```

B. Positional Analysis

```
// Hlavní funkce, která je volána
void apply(WaferMap waferMap)
{
      // Inicializuje mapu pro počet vadných čipů na retiklu
      int reticleMap[][] = new int[RETICLE_HEIGHT][RETICLE_WIDTH];
      fillByZero(reticleMap);

      // Inicializuje mapu pro počet testovatelných čipů na retiklu
      int reticleTestableCount[][] =new int[RETICLE_HEIGHT][RETICLE_WIDTH];
      fillByZero(reticleTestableCount);

      for (int r = 0; r < ROWS; r++)
      {
            for (int c = 0; c < COLLUMNS; c++)
            {
                  if (waferMap.isTestableChip(r, c))
                  {
                        // Jenom testovatelné čipy se počítají

                        // Přepočítáme souřadnice waferu
                        //   na pozici v retiklu.
                        int reticleRow = r % RETICLE_HEIGHT;
                        int reticleCollumn = c % RETICLE_WIDTH;

                        reticleTestableCount[reticleRow][reticleCollumn]++;

                        int bin = waferMap.getBin(r, c);
                        if (isBadBin(bin))
                        {
                              reticleMap[reticleRow][reticleCollumn]++;
                        }
                  }
            }
```

```
	}

	for (int r = 0; r < RETICLE_HEIGHT; r++)
	{
		for (int c = 0; c < RETICLE_WIDTH; c++)
		{
			// Spočítá defektivitu na jedné pozici retiklu.
			double defectivity = 100.0 * reticleMap[r][c]
				/ reticleTestableCount[r][c];
			if (defectivity >= DEFECTIVITY_THRESHOLD)
			{
				// Pokud přesahuje stanovenou mez, označí se čipy,
				//   které se kryjí s pozicí na retiklu, za špatné.
				markReticleChip(waferMap, x, y);
			}
		}
	}
}

// Označí všechny čipy na waferu, které se kryjí s retiklem
void markReticleChip(WaferMap waferMap, int retX, int retY)
{
	for (int r = retY; r < ROWS; r += RETICLE_HEIGHT)
	{
		for (int c = retX; c < COLLUMNS; c += RETICLE_WIDTH)
		{
			waferMap.setBin(MARK_BIN, r, c);
		}
	}
}
```