案件スキルマッチング

ビースラッシュ株式会社

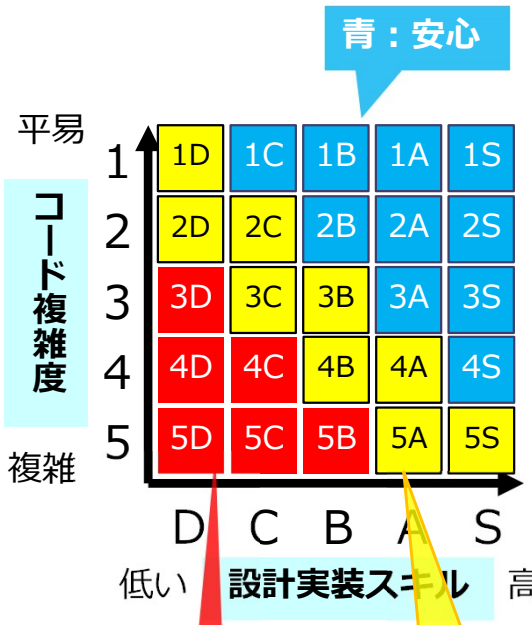
案件のコード複雑度と担当者スキルの組み合わせ

- コード複雑度は、AtScopeのリファクタリングスコアで判定
- 設計実装スキルは、10個の質問と仕様変更への対応方法でセルフ判定

コード複雑度

ソースコードの構造スコアで判定
構造スコアはAtScopeで計測します

コードレベル		構造スコア
1	戦略資産	100～80
2	組織資産	79～40
3	属人資産	39～ 0
4	在庫	-1～ -100
5	債務超過	-101以下

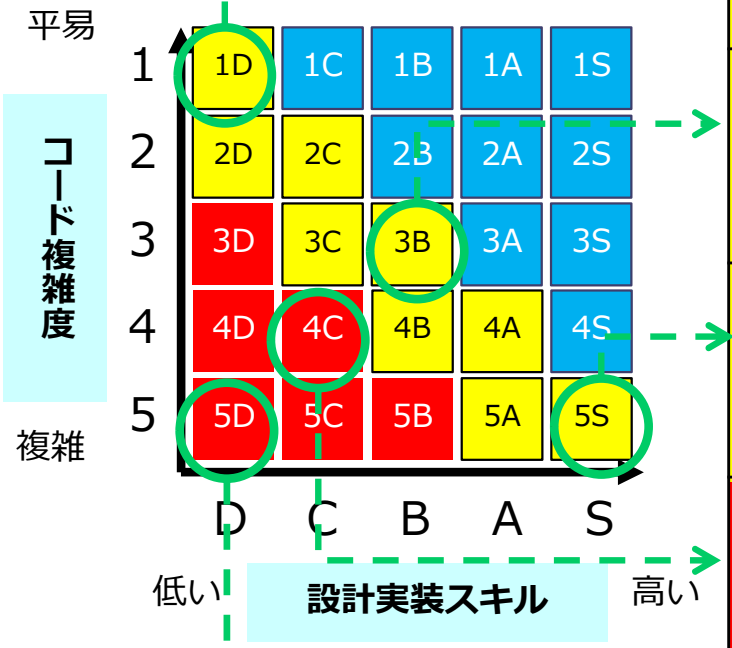


設計実装スキル

設計力と実装力で判定
設計力は、設計に関する10個の質問
実装力は、仕様変更への対応方法で判定します

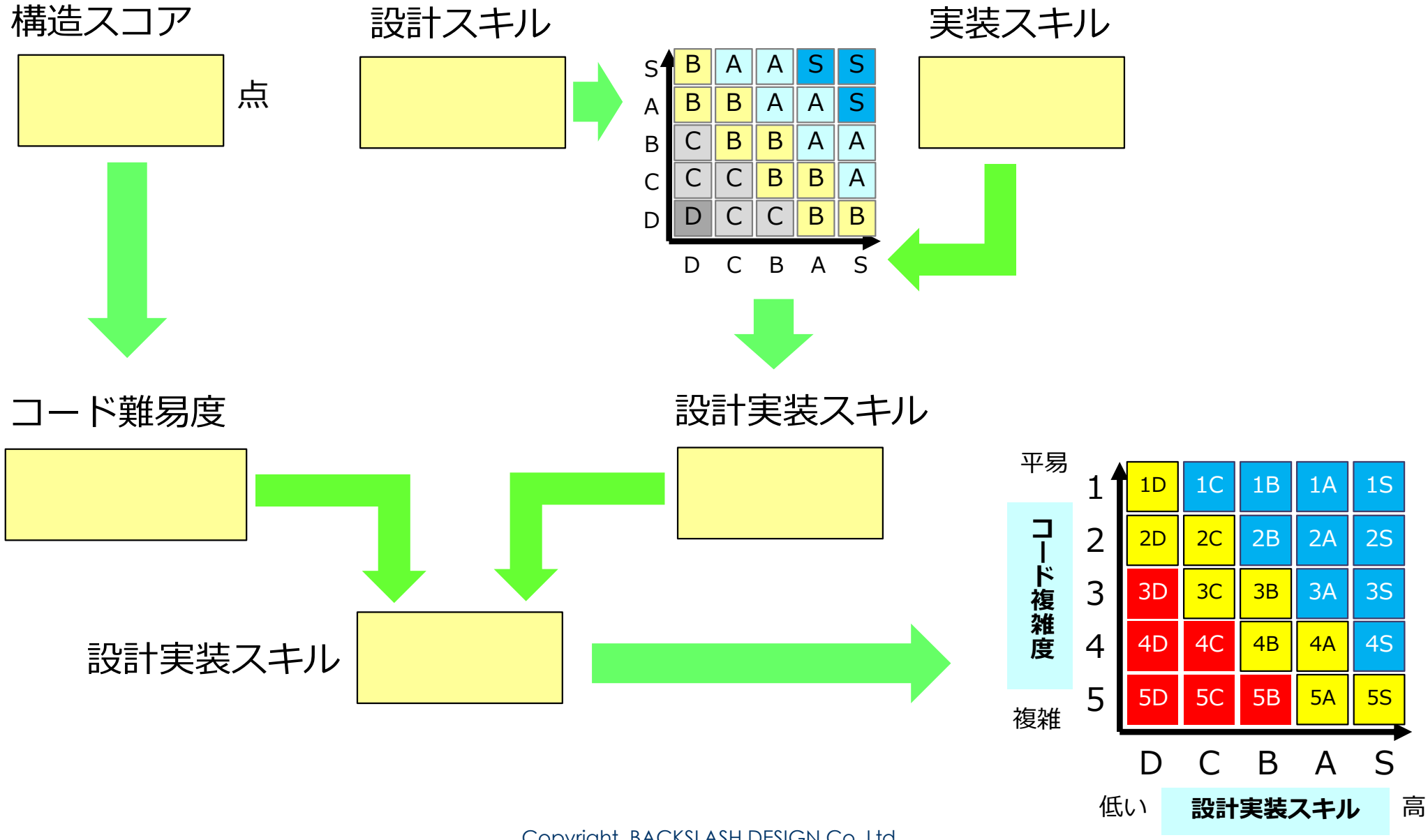
個人スキル		設計力	実装力
S	超級者	9割実践	本質抽象化
A	上級	8割実践	設計意図
B	中級	6割実践	データ構造
C	初級	4割実践	関数化
D	初心者	4割未満	if文追加

マッチングの典型例



	コード複雑度	設計実装スキル	予測される開発状況
1D	戦略資産	初心者	初心者でもスキルアップしながら開発できる。 コードがシンプルなので、それを参考に機能追加できる。指導者が必須。
3B	属人資産	中級	設計レビューすることで、徐々に改善しながら開発できる可能性もある。 マネジメントのバックアップが必要。
5S	債務超過	超級者	重大箇所から改善を重ねる可能性もある。 少なくても、今以上の複雑度にならないよう作る。（もしくは、あきらめて退職）
4C	在庫	初級	動くコードはできるかもしれないが、 不具合のもぐらたたき状態。
5D	債務超過	初心者	無謀なアサイン。 コードはさらに悪化。担当者は疲弊。

マッチング判定シート



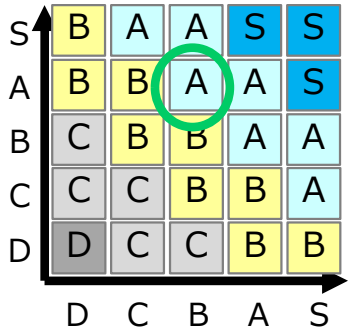
判定例

構造スコア

20 点

設計スキル

A



実装スキル

B

コード難易度

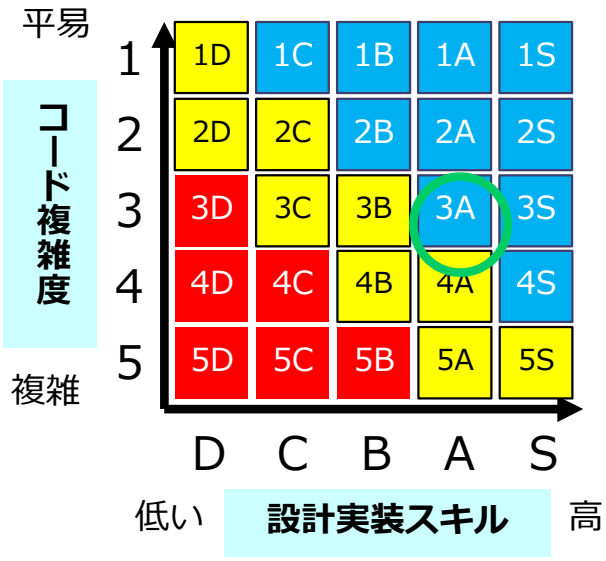
3 (属人資産)

設計実装スキル

A

設計実装スキル

3 A



BACKSLASH DESIGN

案件スキルマッチング
設計力判定

ビースラッシュ株式会社

10個の問いにYes/Noでお答えください

- 質問の内容が分からないときはNoにチェック

No	質問	YES	NO
Q1	変数にWhatの名称を付けていますか？		
Q2	変数はファイル内にカプセル化していますか？		
Q3	関数は、内部処理を一言で表現する名称を付けていますか？		
Q4	関数内のコードブロックごとに、何をしているのかをコメントしていますか？		
Q5	関数が長く（30行超）なったら、複数関数に分けていますか？		
Q6	ファイル名称の用語が重複しないようにしていますか？		
Q7	ファイル内の変数は同一目的になるようにしていますか？		
Q8	ファイルの呼出関係は単方向にしていますか？		
Q9	ファイルが大きく（2000行超）なったら、複数ファイルに分けていますか？		
Q10	エラーや例外発生時の処理ルートを決めていますか？		

設計力判定

レベル		実践割合	
S	超級者	9割実践	大局を見て、局所的な変更を行うことができる。
A	上級	8割実践	ファイル単位の設計構造が徐々に整っていく。
B	中級	6割実践	データ構造と手続きのバランスが整っていく。
C	初級	4割実践	関数が増えていく。 関数構造を考慮していないためスパゲティ化する。
D	初心者	4割未満	局所的修正を重ねてしまい、変更すればするほど複雑化してしまう。何をすればよいかわからない。

BACKSLASH DESIGN

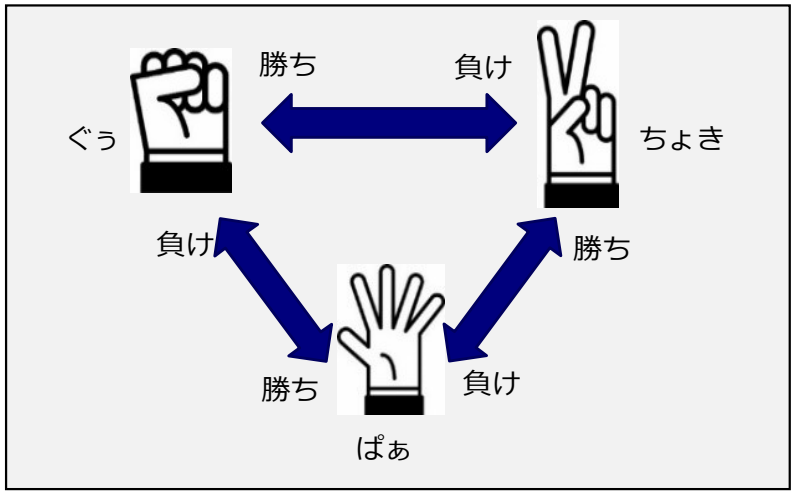
案件スキルマッチング
実装力判定

ビースラッシュ株式会社

実装力判定

- キーボードから自分の手を入力し、
- コンピュータの手と勝負をして、
- 勝ちか負けかを画面に出力します

問い：3人のじゃんけんに
どのように対応しますか？




```
#include <stdio.h>

int main(void)
{
    int player_hand; /* 自分の手 */
    int cpu_hand;    /* 相手の手 */

    /* 自分の手を入力する */
    printf("あなたの手を入力してください。¥n");
    printf(“ 1 =ぐぅ  2 =ちょき  3 =ぱぁ ¥n ");
    scanf("%d", &player_hand);

    /* 相手の手を決める */
    cpu_hand = 2;  /* 相手の手は ちょき とします */

    /* 勝敗を判定し、結果を表示する */
    if (player_hand == 1) {
        /* 自分の手が ぐぅ の場合 */
        if (cpu_hand == 1) {              /* 相手の手が ぐぅ */
                printf("あいこです。¥n");
        } else if (cpu_hand == 2) {       /* 相手の手が ちょき */
                printf("あなたの勝ちです。¥n");
        } else if (cpu_hand == 3) {       /* 相手の手が ぱぁ */
                printf("あなたの負けです。¥n");
        }
    } else if (player_hand == 2) {
        /* 自分の手が ちょき の場合 */
        if (cpu_hand == 1) {              /* 相手の手が ぐぅ */
                printf("あなたの負けです。¥n");
        } else if (cpu_hand == 2) {       /* 相手の手が ちょき */
                printf("あいこです。¥n");
        } else if (cpu_hand == 3) {       /* 相手の手が ぱぁ */
                printf("あなたの勝ちです。¥n");
        }
    } else if (player_hand == 3) {
        /* 自分の手が ぱぁ の場合 */
        if (cpu_hand == 1) {              /* 相手の手が ぐぅ */
                printf("あなたの勝ちです。¥n");
        } else if (cpu_hand == 2) {       /* 相手の手が ちょき */
                printf("あなたの負けです。¥n");
        } else if (cpu_hand == 3) {       /* 相手の手が ぱぁ */
                printf("あいこです。¥n");
        }
    }
    return 0;
}
```

実装力判定

レベル		対応方法	対応内容
S	超級者	本質を抽象化	そもそも、じゃんけんは勝ち負けを決める目的。 勝ち負けの判断ロジックは、手の種類が2種類の時だけ、勝ち負けが決まり、あとはあいこ。
A	上級	構造化で対応	判定ロジック以外も関数化する。 STS構造にすることで変化に対応しやすくなる。
B	中級	データ構造で対応	3人の手の情報を配列化。（決定表） 配列を作れば4人でも5人でも対応しやすい。 判定ロジックはシンプルになる。
C	初級	モジュール化で対応	判定ロジックを関数化することで保守性を上げる。 但し、判定ロジックは複雑なまま。
D	初心者	手続きで対応	今のコードの流れの中でif文対応する。 一筆書きが増長。