はじめに

この本はデザイナーとプログラマーのための本です。しかし一方で、この本はデザインに関する具体的手法やプログラムコードの書き方について書かれた本ではありません。より重要なもの、すなわち皆さんの製品を利用する「人」について書かれたものなのです。

たとえどんなにすばらしい製品であっても、人がそれを使ってくれなくては意味がありません。世界でいちばん美しく、頑丈で、そしてエレガントな絵筆を作っても、その筆で誰も絵を描いてくれなければ、そのために費やした時間は無駄になってしまいます。

人々が使いたくなるようなアプリケーション（アプリ）やウェブサイトを、皆さんが構築するのをお手伝いするのがこの本の目的です。

この本は「テクニックの章」と「アイデアの章」に分かれています。「テクニックの章」では、具体的なテクニック ── デザインの過程において皆さんの製品をより使いやすいものにするための手法 ── を紹介しています。具体的には絵コンテ、ユーザビリティテスト、紙を使ったプロトタイプ作成などです。皆さんが具体的に実行できる事柄、何かのときに思い出してすぐに使えるツールを紹介します。

「アイデアの章」では、より抽象的なアイデアや概念を紹介します。文章の書き方、どの程度リアルなデザインにすればよいのか、アニメーションはどのような場面で使うのがよいか、といった事柄です。「アイデアの章」ではアプリやウェブサイトをデザインする際に、検討すべき事柄を説明します。

テクニックの章

「テクニックの章」の冒頭には歯車のアイコンが置かれています。

「テクニックの章」は基本的には同じ構成になっています。まず、その章で紹介するテクニックが適用できる場面や状況をあげます（すべてのテクニックがすべての場面で使えるわけではありません）。続いてテクニックそのものの内容と、それをどう使うべきかを説明します。多くの章では最後に具体的な例を示します。この例はこの本を通して完成させていく架空のアプリに基づくものです。

以前は「Hello World!」がプログラミング関連の本のいちばん最初の例として登場することが多かったのですが、最近ではTwitter（ツイッター）用のアプリが使われることが多いようですので、この本でもツイッター用アプリを取り上げます。面白くするために、

一般の人が利用するものではなく、ビジネス用のアカウントを複数の人が利用するためのアプリについて検討することにしましょう。このアプリをBizTwitと呼びましょう。

「テクニックの章」は「レシピ」だと考えてください。この本は、1章から順に最後まで読んでももちろんかまいませんが、具体的なトピックに関する章を拾い読みしても役立つはずです。「テクニックの章」は短いものがほとんどで、関連する章やその他の参考文献も紹介してあります。

アイデアの章

「テクニックの章」では特定のテクニックを紹介し、それをどう適用すべきかを説明していますが、「アイデアの章」にはあまり具体的な記述はありません。「アイデアの章」は、さまざまな概念を紹介し、インスピレーションの源としての役割を果たす章です。したがって、具体的なテクニックやルールに関する記述はあまりありません。具体的なテクニックについて触れたり、「テクニックの章」を参照したりする場合もありますが、ほとんどの場合は、より一般的な概念について説明しています。たとえば、「どの程度リアルなデザインにすべきか」「アニメーションをどう使うのがもっとも効果的か」「モードとは何か」「ゲームから何を学べるか」といった内容です。

「アイデアの章」の冒頭には電球のアイコンが置かれています。

「アイデアの章」に紹介されているアイデアは、皆さんが取り組んでいるプロジェクトに必ず適用できるというものではありません。多かれ少なかれ、人の行動は予測が難しいものです。皆さんの製品を使うとき、利用者は必ずしも期待どおりの行動をしてくれません。また、一般的な「法則」に従って行動してくれないときもあります。

ここでユーザーインタフェース（UI）の世界を少し離れて、人々の行動を予期するのがいかに難しいか、その例を紹介しましょう。皆さんは、厚生労働省の下に新たに設けられた、国民の安全と健康増進を担当する部署に配属されたとしましょう。さて、何から手をつけるべきでしょうか。まず、注目したのが自転車事故です。毎年数多くの死傷者が出ています。

調査結果によるとヘルメットをかぶることで、自転車に乗っている人の怪我をだ

いぶ防げることがわかっています。そこで、自転車に乗る人にヘルメット着用を義務化すれば、怪我をする人の数を減らせると考えられます。その結果、国民の安全と健康増進に寄与できる施策となるというわけです。

じつは、「自転車利用者に対するヘルメット義務化法案」はこれまでさまざまな国や地域で導入されましたが、その結果は予想外のものでした。

2009年にオーストラリアのマクオーリー大学のピエト・ド・ヨンが「自転車利用者のヘルメット義務化による健康への影響」と題する論文[†]を発表しました。その研究によると多くの人はヘルメットをかぶるのがとても嫌いで、ヘルメットの着用が義務化されると自転車に乗ること自体をやめてしまうというのです。

この結果から、この論文の著者はヘルメット着用を義務化するとかえって住民の健康を害してしまうことになると結論づけています。義務化により怪我をある程度は防止できるのは確かなのですが、自転車に乗るのをやめてしまう人の多くが自動車に乗って出かけるため、最終的には健康に悪影響を与えてしまうというわけです。

法律を作るまで誰も試してみようとはしませんでした。法律によって影響を受けた人々は、法律を作った人々がまったく考えもしなかった行動をとったのです。

UIのデザインについても同じようなことが起こります。デザインの変更は必ずしも皆さんが意図したとおりの結果をもたらしません。そして、時としてまったく逆の結果が出てしまうこともあるのです。

この本で紹介している「ルール」を実践する際にも、このことを肝に銘じておいてください。ルールに従ってアプリやサイトをデザインしたとしても、結果として利用者が意図しない使い方をしたり、ウェブサイトで「迷子」になってしまったり、とても考えられないような不可解な行動をしたりして、皆さんにとっては当たり前と思われることでもユーザーはできない可能性があるのです。

自分の製品に対して、一連の「ユーザビリティルール」を適用しさえすれば、「有用なアプリやサイトができあがる」とは決して思わないでください。UIをデザインする際には、「常識」を働かせルールに依存しすぎないでください。ルールを覚えましょう。でも、皆さんの製品を良くするのであれば、ルールを破ることを躊躇する必要はありません。筆者が言うとおりにするのではなく、筆者の言葉をインスピレーションの源として利用してください。そして、自分のデザインを必ずテストしてください。

† この論文は、http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1368064で読むことができます。この本に登場するウェブページへのリンクを、この本のサポートページ（http://www.marlin-arms.com/support/d4u/）に用意したので活用してください。

本書の構成

この本の各章は、典型的なデザインの過程で適用できる順番に並んでおり、大きく3つの部分 ── リサーチ、デザイン、インプリメンテーション ── に分かれています。

リサーチ

製品を作るとなると、すぐにデザインに取りかかりたい（プログラマーならばすぐにコードを書き始めたい）と思うかもしれません。確かにそうするのが正解だという場合もあるかもしれません。しかしほとんどの場合、まずは「リサーチ（調査）」から始めるのがよいでしょう。まず「誰のための製品なのか」、そして「どんな問題を解決したいのか」を考えましょう。

デザイン

製品の利用者が抱えている問題を解決する方法を考えましょう。解決策をデザインし、コードを書く前にまずテストしましょう。文書で間違いを正すのは、コードを直すよりもはるかに簡単です。

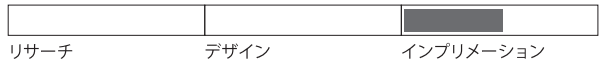
製品を作り上げるという観点からすると、このステージは開発過程においてもっとも重要なものと言ってもよいでしょう。したがって、この本でももっとも多くのページを割いて説明します。

インプリメンテーション

アプリやサイトの開発段階です。しかし、テストし続けてください。以前に仮定したことは正しかったでしょうか。デザインの意図どおりに動いていますか。実際に動かしてみて、どのように使われていますか。インプリメンテーションの出来栄えに満足できますか。エラーはどう扱っていますか。本番のデータはうまく処理できていますか。実行速度は十分ですか。

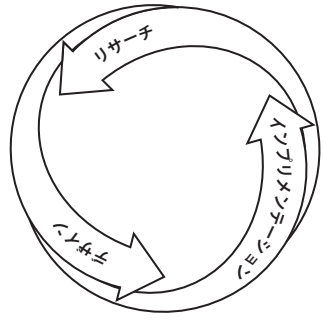
「アイデアの章」を挟む位置は、「理性」というよりは「感性」で決めました。皆さんがいちばん見つけやすそうな場所に置きましたが、アイデアの多くはほとんどの場面で役に立つと思います。「テクニックの章」は「アイデアの章」に比べると、わかりやすい位置にあると思います。

「テクニックの章」の冒頭には、次のような図があります。



網掛けされている部分が、開発過程全体において、その章で説明するテクニックが使われる可能性が高い位置、もっとも重要な意味をもつ位置を示しています。たとえば、上の例は、インプリメンテーションの最初の段階で使われることが多いテクニックであることを示しています。しかし、多くのテクニックはデザイン過程のさまざまな場面で役に立つでしょう。この図は、あくまでも大まかな目安と考えてください。

この図を見ると典型的な開発プロセスが直線的に進むかのように見えてしまいますが、一般にデザインの過程は「反復的」なものです。次の図のように、グルグルと回るもの、繰り返すものだと考えたほうがよいでしょう。



しかし、開発過程は「直線的な作業の繰り返し」とみなすこともできますから、この本ではプログレスバーのような横長の図を用いることにしました。意図を汲み取っていただけたらと思います。

それでは始めましょう！

意見と質問

この本（日本語翻訳版）の内容については、最大限の努力をもって検証および確認していますが、誤りや不正確な点、誤解や混乱を招くような表現、単純な誤植に気づかれることもあるかもしれません。この本を読んで気づいたことは、今後の版で改善できるようにお知らせください。将来の改訂に関する提案なども歓迎します。連絡先を以下に示します。

株式会社オライリー・ジャパン
〒160-0002 東京都新宿区坂町26番地27 インテリジェントプラザビル1F
電話 03-3356-5227
FAX 03-3356-5261
電子メール japan@oreilly.co.jp

この本のウェブページのアドレスは次のとおりです。

http://www.oreilly.co.jp/books/9784873116082/
http://pragprog.com/book/lmuse/designed-for-use/

翻訳者のサイトには簡単な紹介、正誤表や追加情報が掲載されています。

http://www.marlin-arms.com/support/d4u/

オライリーに関するそのほかの情報については、次のオライリーのウェブサイトを参照してください。

http://www.oreilly.co.jp/
http://www.oreilly.com/

謝辞

この本の執筆にあたっては大勢の方々のお力添えをいただきました。お世話になった方のお名前をすべてあげたつもりではありますが、万一漏れておりましたら心からお詫びし、不手際の償いとしてスイスチョコレートを一箱進呈いたします。

不手際といえば、この本に誤りがあった場合、責めを負うべきはもちろん著者の私であり、ここにお名前をあげたすばらしい方々には何の責任もありません。

まず第一に、この本の編集を担当してくださり、私の支離滅裂な原稿を正しい英語に書き直してくださったジル・スタインバーグ氏、ありがとうございました。

また、この本の誤りをしらみつぶしに調べてくださり、見事なフィードバックをくださり、アイデアを提供してくださった査読者の方々にも御礼申し上げます。（順不同で）次の方々です。

エバーノート社のインタラクションデザイナー、キース・ラング氏（ブログ http://UlandUs.com）

Windows Phoneのエクスペリエンスデザイナー、ジョン・ベル氏（個人ブログ http://www.lot23.com、デザインブログ http://www.designdare.com）

オランダのヨット監視ソフト開発会社でグラフィックデザイナー／UIデザイナーとして研修中のマックス・ステーンバーゲン氏（個人ブログ http://facevalue.virb.com、ツイッター http://twitter.com/maxsteenbergen）

詩と短編小説を書いてくださったシャーロット・シモンズ氏。著書に“*The World's Fastest Flower*”［Sim09］があります（http://www.victoria.ac.nz/vup/2008titleinformation/worldsfastestflower.aspx）。

Mac、iPhone、iPadのソフトウェア開発者、ダンカン・ウィルコックス氏（個人ブログ http://duncanwilcox.com）

インタラクションデザイナーであり「恐るべき子供」であるクリス・クラーク氏（愛称Clarko、ウェブサイト http://releasecandidateone.com）

チューリッヒ大学で心理学を専攻するシャーミラ・エガー氏。査読だけでなく調査も手伝ってくださいました。

スイスにあるビジョネー社でクリエイティブディレクターを務めるデイヴィッド・ネフ氏（http://www.wisegamers.chで同氏の記事が読めます）

銀行のリスク管理ツールを作成している熟練開発者、シビル・アレガー氏（あい

にく個人ブログはないそうです）

ナムコムソフトウェアのAppway担当上級マネジャー、マイケル・トラマー氏（LinkedInのURLはhttp://ch.linkedin.com/in/trummer。http://twitter.com/DrummeratWorkで、ビジネスアプリケーションの開発に関してつぶやいています）。

このほか、26章に掲載されている「Replica Islandでキャラクターが死ぬ場所を示す頻度分布図」の転載を許可してくださったクリス・プルーエット氏（http://replicaisland.blogspot.com）、19章にある「受動的な遅延確認」モデルの使用を許可してくださったクレイトン・ミラー氏（http://rclayton.net）にも御礼申し上げます。

エイマデ・フライズ氏（http://zuendung.ch）は、2枚の車の写真の転載を許可してくださいました。

さらに、1章の携帯ゲーム機「アタリ・リンクス」の開発について筆者の質問に答えてくださった共同設計者のRJ・マイカル氏（http://www.mical.org）と、DrawItの機能を削除したときの経験（25章）を話してくださったボヘミアンコーディング（http://www.bohemiancoding.com）のピーター・オンブリー氏にも御礼申し上げます。

この本のために行った心理学の研究に手を貸してくださったアマンダ・キーファー氏にも感謝申し上げたいと思います。

このほか、大勢の方が著書やブログからの引用を許可してくださいました。こうした方々に御礼申し上げます。

また、ツイッターの写真、名前、ツイート内容の転載や引用を許可してくださった多くの方々にも御礼申し上げます（アルファベット順で、7WP、aidanhornsby、AlphabUX、CoryCalifornia、designdare、fetjuel、gauravmishr、ienjoy、jonbell、larryv、lorentey、louije、maxsteenbergen、neave、oliverw、shawnblanc、thibautsailly、timeboxed、workjonの方々です）。

最後に、この本に掲載した写真の一部を撮影してくださったダマリス・キーファー氏、ありがとうございました（同氏が写っているものも2、3枚あります）。

目次 TABLE OF CONTENTS

賞賛の声 vii
はじめに ix

I部 リサーチ 1

1章 ユーザーリサーチ 3
2章 ジョブシャドウイングとコンテキストインタビュー 7
2.1 ユーザーの観察 7
2.2 ジョブシャドウイング 8
2.3 コンテキストインタビュー 8
2.4 リモートのシャドウイング 9
2.5 コンテキストインタビューの限界 10
3章 ペルソナ 13
3.1 ペルソナのもつ危険性 14
3.2 ペルソナの作成 15
3.3 ペルソナの活用 16
3.4 ペルソナはユーザーリサーチの代わりにはならない 17
4章 活動中心のデザイン 21
5章 付随資料の作成 25
5.1 マニュアル 25
5.2 ブログ投稿 26
5.3 スクリーンキャスト 27
5.4 プレスリリース 27
5.5 「何ができるか」を語る 28
6章 文章のユーザビリティ 31
6.1 なぜ言葉が大切か 31
6.2 文章は読まれない 32
6.3 言葉少なに 33
6.4 ザッと読める文章を書く 34
6.5 文章の無駄は省く 34
6.6 誤解の余地がない文を書く 35
6.7 文章は親しみやすい語り口で 36

6.8 文章の要点を図解する 37
6.9 皆が理解できる言葉を使う 38
6.10 文章をテストする 39
6.11 文章は判読しやすい形で表示する 40
7章 インタフェースデザインと階層 43
7.1 階層構造の視覚的構築 44
8章 カードソート 49
8.1 階層の設計 49
8.2 カードソートの準備 51
8.3 参加者 52
8.4 カードソートの実施 53
8.5 リモートのカードソート 56
8.6 結果の評価 56
8.7 有用な階層を構築するためのガイドライン 57
9章 メンタルモデル 63
9.1 ユーザーの期待 63
9.2 3つのモデル 65
9.3 インプリメンテーションの詳細の隠蔽 67
9.4 メンタルモデルの弱点 69
9.5 メンタルモデルに基づくデザイン 70

II部 デザイン 81

10章 スケッチとプロトタイプ 83
10.1 全体の構造 83
10.2 フローダイアグラム 84
10.3 絵コンテ 84
10.4 スケッチ 85
10.5 ワイヤーフレーム 87
10.6 モックアップ 88
10.7 ツール 90
11章 ペーパープロトタイプテスト 93
11.1 ペーパープロトタイプによるゲリラテスト 94

11.2 ペーパープロトタイプによるフルユーザビリティテスト 96
12章 リアリズム 111
12.1 シンボル 112
12.2 実物のバーチャル版 114
12.3 物理的な制約の再現 117
13章 ナチュラルUI 121
13.1 魔法のジェスチャーを使わない 121
13.2 ジェスチャーの認識 123
13.3 偶発的な入力 125
13.4 取り消し機能 126
14章 フィッツの法則 129
14.1 画面の端には無限の幅がある 130
14.2 パイ型のコンテキストメニュー 131
14.3 小さい対象物同士は隙間を空けて配置する 135
14.4 対象物は小さいほうがよいこともある 135
15章 アニメーション 137
15.1 画面が切り替わったことを知らせる 137
15.2 注意を促す 138
15.3 重要でないアニメーションは避ける 140
15.4 正しいメンタルモデルの構築を助ける 141
15.5 アニメ映画を参考に 143
16章 一貫性 149
16.1 一貫性のなさとは 149
16.2 動作に一貫性をもたせる 150
17章 発見可能性 153
17.1 何を目立たせるか 153
17.2 いつ発見可能にするか 155
17.3 どのように発見可能にするか 156
18章 集中を妨げない 159
18.1 ユーザーの代わりに決定する 159
18.2 意思確認は最初に済ませる 161
18.3 緊急性がなければユーザーの意思決定を促してはならない 162

19章 取り消し機能 165
19.1 動作の取り消しを可能にする 166
19.2 一時的な取り消し 167
20章 モード 169
20.1 自明でないモード 169
20.2 予期されなかったモード 174
20.3 解除法の不明なモード 175
20.4 モードは必ずしも悪ではない 176
20.5 疑似モード 176
21章 プレファレンスよりオプションを 179
21.1 プレファレンスがよくない理由 181
21.2 プレファレンスをなくす方法 182
21.3 プレファレンスをなくせない場合 184
22章 階層化、空間、時間、外界の認知 187
22.1 階層化 187
22.2 空間 188
22.3 時間 191
22.4 よりよい階層化システム 193
23章 スピード 197
23.1 応答速度 197
23.2 進捗状況のフィードバック 198
23.3 スピード感 200
23.4 速度制限 201
24章 機能の抑制 203
24.1 ユーザーの目標を忘れない 204
24.2 5つのなぜ 204
24.3 新機能を追加せず既存の機能を改良する 206
24.4 ひとつの変更で複数の問題解決を 206
24.5 コストを考慮 207
24.6 不可視化する 207
24.7 APIとプラグイン構造を提供する 207
24.8 ユーザーの言葉に耳を傾ける 208

24.9 ユーザーの意見を聞きすぎない 209
24.10 すべてのユーザーが皆さんのユーザーとはかぎらない 210
25章 機能の削除 213
25.1 調査の実施 213
25.2 ユーザーへの情報提供 215
25.3 代替手段の提供 215
25.4 製品は自分たちのもの 216
26章 テレビゲームに学ぶ 219
26.1 何が面白いのか 219
26.2 製品がゲームと異なる理由 221
26.3 ゲームから学べること 224
26.4 面白い vs. 使える 230

III部 インプリメンテーション 233

27章 ゲリラユーザビリティテスト 235
27.1 テストの頻度 236
27.2 テストの準備 236
27.3 テスターの探し方 237
27.4 テスターの人数 237
27.5 テストの実施 237
27.6 結果 238
28章 ユーザビリティテスト 241
28.1 ユーザビリティテストは本格的でなくてよい 241
28.2 テストの頻度 243
28.3 テスターの人数 243
28.4 誰が製品をテストするべきか？ 245
28.5 テスターの探し方 246
28.6 テストの種類 247
28.7 テストの準備 248
28.8 テストの実施 249
29章 対面テスト 251
29.1 テストの実施 251

30章 リモートテスト 259
30.1 調査者介入型のリモートテスト 259
30.2 調査者不在型のリモートテスト 269
31章 ユーザビリティテスト「べからず集」 271
31.1 ユーザーインタフェースに表示される語句は使わない 271
31.2 テスターに影響を与えない 272
31.3 ストレスのかかる状況を避ける 273
32章 ユーザーエラーはデザインエラー 275
32.1 エラーメッセージでユーザーを責めない 275
32.2 エラーをなくす 278
33章 A/Bテスト 283
33.1 いつA/Bテストを行うか 283
33.2 成功とは何か？ 285
33.3 テストの準備 286
33.4 テストの実施 286
33.5 テスト結果の解釈 287
33.6 留意点 288
34章 利用データの収集 291
34.1 速度の測定 291
34.2 出口点 292
34.3 不具合の記録 292
34.4 ユーザーの行動 293
35章 ユーザーからのフィードバック 295
35.1 思いもよらない使われ方 295
35.2 たちの悪いフィードバック 296

最後に ── でも、まだ終わってはいません 299
訳者あとがき 300
参考文献一覧 302
索引 304

I部 リサーチ

第 I 部のテーマは「リサーチ」、すなわち事前の調査です。なぜリサーチが肝心なのか、どのようなリサーチが効果的なのか、また何を避けるべきかを理解しましょう。そして人の行動を観察する術とインタビューのしかたを学びます。目的に沿った形でリサーチを進め、製品のデザインをくっきりと浮かび上がらせるためには、どのように「ペルソナ」を利用すればよいでしょうか。それが飲み込めたら、「カードソート」を用いて製品を組み立てる方法を見ていきます。
ではまず、なぜリサーチが肝心なのかを探りましょう。

1章
ユーザーリサーチ

デザイナーが自分自身のデザインについて語るとき、「人間中心の」または「ユーザー中心の」といった言葉をよく使います。ずいぶん曖昧な言い方ですが、「自分の製品を使うことになる人々を常に念頭に置き、その人々のために最良の製品を生み出そうと努めている」という意味で使われています。

さて、人間中心、ユーザー中心のデザインを実現するためには、具体的に何をどうすればよいのでしょうか？

この問いに答えるのは思いのほか難しいのですが、その答えのひとつの鍵となるのがユーザーリサーチなのです。

「ユーザーの目的は何か」を調べるには、あるいは「どうすればユーザーをその目的へ導けるのか」を調べるにはどうすればよいのでしょうか。「尋ねてみる」というのがいちばんわかりやすい答えかもしれません。確かにこの方法で有益な情報が得られることもありますが、そうして得られた意見を吟味する際には注意が必要です。

フォード・モーターの創設者ヘンリー・フォードは「人々に何が欲しいかと尋ねていたら、『もっと足の速い馬だ』という答えが返ってきていただろう」と言ったそうです。なぜ、このような答えが役に立たないのでしょうか。普通の人は、デザイナーや開発者のようにじっくり考えたりはしてくれないのです。自分たちが解決したい問題の本質は何か、どんな製品ならその問題を根本的に解決してくれるのか、そういったことを考えてはくれません。いつも馬の世話をしなければならないのは「当たり前だ」と思っていて「解決しなければならない問題だ」とは思っていませんし、「馬小屋を建てればいい」「馬の世話係を雇えばいい」といったように、その場しのぎの対処法を考え、問題の本質を見ようとはしないのです。「本質的な問題を解決してくれるもの」ではなく、「今までの状況を少しばかり改善してくれるもの」を求めるのです。

そのため、ほとんどの人は「どうすれば（how）」自分たちが抱えている問題を解決できるのかをデザイナーや開発者に対して伝えることができないのです。さらに言えば、「何（what）」が問題なのかを伝えることすらできない人も大勢います。そして何より困るのは、我々が「こんな製品ならどうでしょう」と提案しても、その製品が問題を解決してくれるものになるのか想像して判断することすらできない場合も多いのです。

「アタリ・リンクス」というアタリコープ社の携帯ゲーム機を例にして説明しま

しょう。1990年代初め、米国の携帯ゲーム機市場の主役は任天堂の「ゲームボーイ」でした。

ほとんどの子供がゲームボーイを持っており、持っていない子はこの小さな灰色のゲーム機を誕生日に買ってもらえるのを心待ちにしていました。競合するゲーム機器企業のアタリはそうした市場への参入を目指したのです。

アタリは複数の「フォーカスグループ」から話を聞いたあと、ゲームボーイよりはるかに高性能な製品を投入することに決めました。「リンクス」と名づけられたこのゲーム機は、カラー画面と高速プロセッサを搭載し、性能面ではゲームボーイをはるかに凌いでいました。アタリはゲーム機本体もかなり大きくしました。フォーカスグループの人々が、大型のゲーム機のほうが好きだと言ったからです。

結果、このゲーム機は大はずれ。誰もリンクスを欲しがりませんでした。

リンクスの共同設計者であるロバート・J・マイケル[†]にこの件について尋ねると、次のように話してくれました。

† http://www.mical.org

> 私がリンクスから学んだもっとも貴重な教訓は、「フォーカスグループの意見を鵜呑みにしてはならない」です。私たちはリンクスについて、特に本体のサイズと形をめぐってフォーカスグループを使ったテストを何度も行いました。異なるモデルをいくつも提示し、「どちらが好きですか？ どれがいちばんいいと思いますか？」と尋ねました。大きなものと小さなもの、巨大なものと極小のものを見せたのです。何度テストを繰り返しても大きなほうが選ばれました。誰もが「大きなのがいい！ 大きくして！ 払った金額に見合うものを手に入れたと実感したいから」と言いました。そういうことならと思い、私たちは大きなものを作りました。そしてリンクスが世に出ると、突如、誰もが「こんなに大きいの？ なんで〜」と言ったのです。ひどい話ですよ。最初のリンクスは筐体の中が隙間だらけだったんです。自分たちの直感に従うべきでした。フォーカスグループの言うとおりにしてしまったのが、大間違いだったのです。

フォーカスグループ

ユーザーリサーチのひとつの手法として「フォーカスグループ」を対象としたインタビューがあります。新しいサービスや製品、類似品などを特定のグループに見せたり使ってもらったりして、主観的な反応と意見を記録します。こうしたデータをもとにその製品に対する一般の人々の反応を予測するのです。こうした手法自体を「フォーカスグループ」と呼ぶ場合もあります。

結局、人々は携帯ゲーム機に何を求めているのか実は自分でもわかっていない、ということがわかりました。それをどう使うつもりかを正しく予測する能力に欠けていたのです。人々はその主張とは裏腹に、大型で高性能のゲーム機など求めてはいませんでした。むしろ子供たちはゲーム機をかばんに入れて持ち歩きたかったのです。そうするにはリンクスは大きすぎた上、高性能のプロセッサとカラー画面のために電池は4時間足らずで切れてしまいました。ということは真新しい電池をセットしても、学校へ行く日ですら1日ももたないのです。おまけに、高性能のハードウェアを揃えたためにコストがかさみ、多くの親は子供に気楽に買い与える気にはなりませんでした。

高性能のプロセッサとカラー画面を搭載した大型のゲーム機を求めたのは、そのようなゲーム機でプレイをしたらどんなにすばらしいだろうと想像を膨らませたからです。現実に求めていたのは、簡単に持ち運べて電池を交換しなくても長時間プレイできる、安価で小型のゲーム機でした。

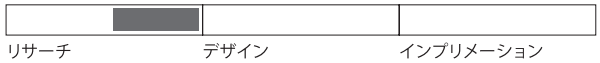
アタリは慌てて小型版を発売しましたが、それが市場に出回る頃には手遅れになっていました。リンクスの販売数はわずか50万台。一方、任天堂のゲームボーイはその数を1億2,000万台近くまで伸ばしました。

現時点で、我々には製品の対象者が誰になるかはわかっています。しかしその対象者には「自分自身の欲しいもの」はわかってはいないのです。単に「何が必要ですか」と尋ねるわけにはいかないのです。それは自分たちで見つけ出す必要があるのです。

問題を見つける	解決策を見つける
人々が今、（特に意識せずに）していることを見つけ出す	人々がすでにしていることをより簡単で効率的に行える方法を見つける
人々が本当はしたくないが、しなければならないことを見つけ出す	人々がしたくないことをせずに済む方法、あるいは少なくともそうしたことを、より楽しくできる方法を見つける
人々がしたいと思っていることを見つけ出す	人々がしたいことができるようにする方法を見つける

2章以降を読めば、その方法がわかるはずです。

8章 カードソート



概要

カードソート（カード分類）は、サイトやアプリなどの「パーツ」をカードに書き出して、そのカードを使ってユーザーにどう提示するか、ほかのパーツとどう関連づけるかを検討するものです。作ろうとしているものがある程度複雑で、さまざまな機能や部品を分類したり並び替えたりする必要がある場合に有効な方法です。

たとえば、皆さんがウェブサイトを設計していて、個々のページをどう構成しようかと悩んでいる場合や、たくさんの機能をもつアプリの開発に取りかかろうとしていてどうやって機能を整理したらよいかわからないといった場合に、カードソートがとても役に立つでしょう。

なぜ役に立つのか

ウェブサイトやアプリなどデザインの対象となる製品のとらえ方は、開発者側から見た場合とユーザー側から見た場合とで大きく異なります。例として、ある会社のウェブサイトを作ろうとしている場面を考えてみましょう。開発者のほうはその会社の部門構成などをよく知っていますから、サイトの構成を考える際にはどうしてもそうした「実体」に影響を受けることになります。一方ウェブサイトを訪問する立場からすると、会社内の事情などまったくわかりません。したがって、社内の者にとっては「至極当然」と思われることでもユーザーは「わけがわからない」という反応をする可能性さえあるのです。

カードソートによって、一般の人々の見方を発見できる可能性が高くなります。

前提条件

ありません。

8.1 | 階層の設計

7章で多くのウェブサイトやアプリが階層構造を利用していることを紹介しました。この章でもまずウェブサイトの例を見てみましょう。

たとえば、ノートパソコンの内蔵カメラが動かなくなってしまったとします。そこでメーカーのウェブページをブラウザで開きます。大きなメーカーならば、部門ごとのタブが表示されている場合が多いでしょう。まず、「コンピュータ」のタブを

クリックして、「ノートパソコン」をクリックします。「シリーズ」の一覧があるので、その中から自分のパソコンと同じものを見つけます。するとそのシリーズに属するパソコンの「型番」が並んでいるので、該当するものを見つけてまたクリックします。同じ型番のパソコンに関するページが表示されたので、そのページの「サポート」欄をクリックして、カメラ関連の故障に関する情報（や解決策）がないかどうかを確認します。

ここで、皆さんがたどった道筋を書いてみましょう。

コンピュータ → ノートパソコン → シリーズ → 型番 → サポート → 解決策

しかし、同じページに到着する方法はこれだけではありません。まずこの会社のウェブサイトの「サポート」のページに行く人もいるかもしれません。また、ほかの人はトップページでいきなり型番を入力して検索をするかもしれません。

ウェブサイトをデザインする際に、訪問者がどのような道をたどるかを知るにはどうすればよいのでしょうか？ 訪問者はどのようにすれば目的のページが見つかると思っているのでしょうか？

8.1.1 カードソート

こうした質問に対する解答を見つける方法としてよく用いられるのが「カードソート」で、一般の人が「どこに何があるのがよいと思っているのか」を見つけ出すのにとても役に立つ手法です。カードソートは本当に簡単です。ドナ・スペンサーは著書 "*Card Sorting: Designing Usable Categories*" [Spe09] の中で次のように説明しています ── 「肝となるのは、とても簡単なテクニックです。カードに項目を書いて、ほかの人にカードをグループ分けしてもらうように頼むだけでよいのです」。

では、具体的な方法を見ていきましょう。カードソートはさまざまな場面で利用できますが、ここではウェブページに表示するタブやメニューの項目を決めたり、アプリのひとつの画面に表示するボタンの種類を決めたりする場合を考えてみます。

8.2 | カードソートの準備

まず、一束のカード†を用意します。大きさは適当でかまいませんが、慣れるとお気に入りのものができるはずです。

カードの1枚1枚に、並べたり組織化したりする対象のものをひとつずつ書き出します。ウェブサイトを構築するのにカードソートを利用しようとするならば、「個々のページ」あるいは「まとまりとなる一群のページ」をカードの1枚1枚に書き出します。アプリを作成しようとしているのならば、たとえば1画面に表示される各UI要素（ボタンなど）をカード1枚ごとにひとつずつ書き出していきます。

カードに書き出す際には同じレベルにあるものを対象にします。階層の上のほうに位置する項目と、下位の項目とをごちゃ混ぜにしないようにしましょう。必要ならば、上のほうの項目についてカードソートを行ったあとでレベルを一段下げて再度行うといったように、各レベルでカードソートを行います。

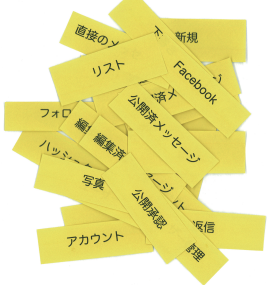
† 最初は「情報カード」あるいは「インデックスカード」などと呼ばれるものが便利でしょう。同じ内容の書かれたカードを何度も使うことになりそうならば、大きめの紙に印刷してそれをカットするほうがよいでしょう。

カードに書く際には、明解で簡単に理解できる語句を使うようにしてください。専門用語や業界用語を使わざるを得ない場合は、カードソートの実行前にその意味を説明し、参加者が自分たちの表現に置き換えられるようにしましょう。

カードソートに使う語句については、表面的な類似性で簡単に分類できてしまうような書き方は避けるようにしてください。同じ文字が使われているとか、音が似ているといった類似性で分類されてしまうのはまずいのです。あくまでも分類の基準は「意味」です[†]。

カードの枚数は20枚から80枚程度になるようにします。20枚に達しないようならば、もっと時間をかけて、まだ表に現れていない（カードに書かれていない）ものを掘り出しましょう。逆に80枚を超えてしまうようならば、参加者は数の多さに圧倒されてしまうことになります。

また、何も書いてないカードも何枚か用意してください。



参加者がカードを加えたり、内容を変更したりすることもありますし、カードソートを複数回することもあるでしょうから、同じカードを何度も手で書くのは時間がかかるのでお勧めできません。大きめの紙に印刷して、それをカットするほうがよいでしょう。

8.3 | 参加者

参加者はひとりだけもよいでしょうが、一度に複数の人に参加してもらうこともできます。どちらもそれぞれの長所があります。複数の参加者に集まってもらうためにはスケジュールの調整が必要ですし、結局は参加者のうちのひとりが全体をリードすることにはなるでしょう。しかし、複数の人が議論しながら分類すること

† ヤコブ・ニールセンは、この問題についての解決策をウェブページで提案しています。http://www.useit.com/alertbox/word-matching.html

で、表には出てこないことの多いユーザーの心の中を知るチャンスとなるかもしれません。

一度に複数の人に参加してもらう場合、参加者の数は増やしすぎないほうがよいでしょう。収拾がつかなくなる事態を避けるには、3人か4人が限度でしょう。

全部で何回ぐらいカードソートをすればよいのでしょうか？ ヤコブ・ニールセンは15人がひとり1回ずつ、つまり合計15回を推奨しています[†]。カードソートは難しい作業ではないので、多くの人にやってもらってもたいして時間はかかりません。人によって分け方はさまざまですから、カードソートをやればやるほど全体像がよりクリアになってきます。

† http://www.useit.com/alertbox/20040719.html

8.4 | カードソートの実施

参加者が集まったら、どのように実施するか説明します。たとえば次のような感じです。

こんにちは。ルーカス・マティスです。現在、新しいツイッターアプリの開発を計画しています。ご存じの方も多いと思いますが、ツイッターは、いわゆる「ソーシャルネットワークサービス」のひとつです。

私は今、その新しいアプリの全体的な構成について検討しています。どんな画面を用意して、どの機能をどこに置くかといったことです。これから皆さんに「カードソート」という作業をしていただいて、どの機能をどこに置くのが皆さんにとって使いやすいのかを見つけ出すお手伝いをしていただきたいのです。

これからカードをお配りします。そのカードの1枚1枚に語句が書かれています。そして、その語句は今検討中のアプリの中に含まれている何らかの要素を表しています。たとえば、アプリのひとつの機能とか、アプリ中に表示されるウィンドウとかボタンとかそういったものです。ちなみに、こういったもののことを、我々の用語で「オブジェクト」と言います。

これから皆さんに、このカードを分類していくつかの山に分けていただきます。それぞれの山には、似ているものを皆さんの考えでまとめてください。もう少し具体的にご説明しましょう。皆さんに見つけていただきたいのは、表面的な類似性ではありません。たとえば、同じ語で始まるから同じグループに分類するとか

いうことではありません。このアプリを使っているところを想像しながらやってください。たとえば、「どれとどれが同じ画面に表示されるべきか」といったことを頼りにまとめてください。ひとつの画面にいくつかの項目が並べられているとしたら、どれとどれが隣に並ぶのがいいでしょうか。そういったことを手がかりに分類していただきたいのです。

どのグループにも入らないとか、このアプリには入れるべきではないとかいったカードがあれば、そういったカードだけを別によけてひとつの山にしてください。何も書いてないカードとマーカーも用意してありますから、カードに書かれている語句の意味が明解でないと思ったら、その語句を×印で消してしまって、代わりにほかの語句を書いたカードを使ってください。複数の山に入るべきカードがあると思ったときは、同じ語句を新しいカードに書いて、複数の山に入れていただいてもかまいません。

では始める前に、カードを1枚1枚見ながら簡単に説明をしていきます。

次に参加者と一緒に1枚1枚カードを見ていきます。各カードに書かれている語句が、サイトやアプリなどに関連してどんな意味をもつのか、参加者がしっかり理解していることを確認しながら進行してください。

それでは本番です。参加者にカードを分類してもらってください。意味的なまとまりの山に分けてもらいます。

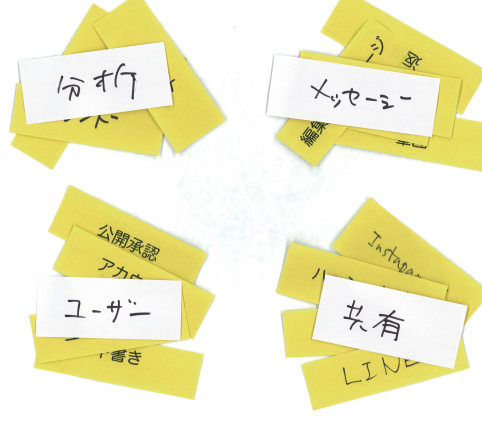
声に出しながら考えてもらったり、自由に質問してもらったりしてください。依頼者側ではこの段階からメモを取り始めます。参加者が新しい語句を思いついたら、新しいカードを追加するか、既存のカードに付け加えます。

参加者が枚数の少ない山を数多く作る傾向があるようならば、できるだけまとめるように促してください。逆に、山の数が少ないのならば、もう少し山を増やすよう促します。

状況によっては、あらかじめ山の分類（と数）を指定しておく方法もあります。この手法を「閉じたカードソート」と呼びます。

どの山にも属さない「仲間はずれ」のカードは、別のところにまとめてもらうようにすることも大切です。こうしたカードは、開発中のアプリあるいはウェブサイトに含める必要のない機能や情報を意味する場合が多いでしょう。

次に、参加者にそれぞれの山にふさわしい名前を付けてもらうよう依頼し、いちばん上にその名前を書いたカードを置いてもらいます（できればこのためのカードの色は変えたほうがよいでしょう）。



山の数が多いときは、できるだけ似通った山を近くに配置するよう依頼してください。参加者の様子やカードに書いた語句の種類によっては、関連する山と山の間に線を引いてもらってもよいでしょう（最初からこれを計画している場合、模造紙のような大きな紙の上に山を作ってもらうとよいでしょう）。

最後にまとめをします。もっとも手っ取り早い方法は、全体を写真に撮ることです。それぞれの山はゴムで束ねて散らばらないようにしておきましょう。

8.5 | リモートのカードソート

カードソートの参加者にオフィスに来てもらう必要は必ずしもありません。都合のよい場所でやってもらうこともできます。websort.netやOptimalSort†といったサイトでカードソートを行うこともできます。

† http://www.optimalworkshop.com/optimalsort.htm

リモートのカードソートでは参加者の数が増えるため、より細かなデータが得られます。ただし、逆に得られなくなる情報もあります。たとえば、どんな語句が混乱のもとになったか、同じことを表すのに参加者がどんな同義語や類義語を使ったか、カードに書かれている語句からどんなことを連想したかといった情報は「集合版」のカードソートでしか得られません。

「集合版」と「リモート版」の両方のカードソートをしてもらうのもよいでしょう。

8.6 | 結果の評価

あらかじめ分類を決めてあった場合（閉じたカードソート）、結果の評価は単純です。次のようにカードごとにどのグループに分類されたかを数えれば、一般の人がそのカードの内容がどこにあるべきだと考えているかがだいたいわかるでしょう。

	グループ1	グループ2	グループ3	グループ4	グループ5	グループ6	グループ7			
カード1		4			3					
カード2			2	6						
カード3	7									
カード4			5	1		1				
カード5			5	4						
カード6	1	4					3			

一方、参加者が自分たちで分類する方式をとった場合、まずグループを定義することから始めてください。人によってグループの名前の付け方は変わりますが、同じ意味合いのグループはまとめるようにしたほうがよいかもしれません。時として、

まったく違う分類をする人もいるかもしれません。そういったことが起こった場合は、同じ情報にアクセスするのに複数の方法を提供したほうがよいのかもしれません。利用するグループを抽出し終わったら、閉じたカードソートの場合と同様に、各カードがどのグループに何回登場したかを数えます。

さあ、これで準備完了です。集まった情報に基づいて階層構造を決定し、レイアウトや情報アーキテクチャにこの階層構造を利用します。統計的な評価はしないほうがよいでしょう。というのは、おそらくそういった評価には十分な情報は集まらないと思えるからです。カードソートは、皆さんがデザインを決定する際の参考とはなりますが、特定の解決策が「正しい」ということの証明にはなりません。

集めた情報を考慮に入れて、アプリの絵コンテ（ストーリーボード）を作ったり、ウェブサイトのページの階層を考えたり、個々の画面要素の階層を決定したりしてください。カードソートから得られるのは、ほかの人が皆さんのアプリやサイトに対してどう考えるかに関する洞察です。ほかの人のメンタルモデル ── 製品がどのように動作すると思っているか ── を推測するもとになるものです。

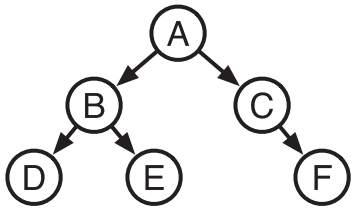
階層構造を決めたら、紙を使ったプロトタイプを作って、簡単なユーザビリティテストをいくつか行ってみるのがよいでしょう（これについては11章で説明します）。これにより、カードソートの結果の解釈が正しかったかどうかを確認できます。

8.7｜有用な階層を構築するためのガイドライン

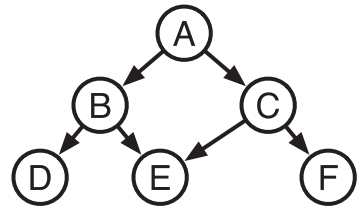
カードソートで得られたデータは階層構造をうまく構築するために役立ちます。その際に役に立ちそうなヒントをいくつかあげておきましょう。

8.7.1　ひとつのものが複数の場所にあってもかまわない

階層を必ずしも排他的に構築する必要はありません。ある要素が、階層中に複数回現れてもかまいません。次の図（右側）のように、ユーザーが同じ結果に至る複数のパスを与えてもよいのです（このような階層構造を「多重階層」と呼ぶ場合があります）。



普通の階層構造



多重階層（Eに至るパスが複数ある）

先ほどノートパソコンの内蔵カメラが故障したときの例をあげましたが、目的が同じでも自分の欲しい情報に到達する経路はユーザーによってさまざまです。製品の分類を順にたどって自分のもつパソコンのモデルのページに到達する人もいれば、「サポート」のページから自分のモデルを探す人もいます。また、「サポート」のページで症状から検索する人もいるかもしれません。こういった人々が自分の行動を想定しながらカードソートを行えば、同じページに到達するのにさまざまなパスが存在する多重階層の構造ができあがることになるでしょう。

8.7.2 階層の深さ

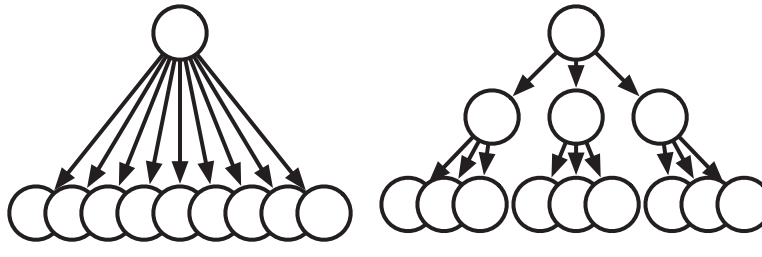
ユーザーインタフェース（UI）の評価の際にクリック数が用いられることがよくあります。ユーザーがゴールに到達するのにクリックは何回までならば許されるのでしょうか？ クリック数が少なければ少ないほどよいのでしょうか？

クリックを少なくするためには、階層をできるだけ浅くしておいたほうがよいでしょう。クリックを2、3回するだけで、ゴールに到達できるようにするのです。しかし、筆者はこうした手法には反対です。なぜかというと、階層を浅くするということは、各レベルで考慮しなければならない選択肢の数が増えてしまうことになるからです。階層のレベルを少なくすれば、横の広がりを増やさざるを得なくなるのです。

クリック数を少なくすることがよい結果を生むわけではないという研究結果[†]も実際にあります。この結果によると、「クリック数はユーザーの満足度にも目的の達成率にも影響がなかった。クリック数が少ないからといって嬉しいというわけでもないし、素早く目的のものを見つけられたとも感じられない」ということが明らかになっています。

† http://uxmyths.com/post/654026581/myth-all-pages-should-be-accessible-in-3-clicksを参照。スーザン・ワインチェンクの『インタフェースデザインの心理学』（オライリー・ジャパン）にも同様の研究が紹介されています。ただし、テレサ・ニールの『モバイルデザインパターン』（オライリー・ジャパン）によると、モバイルアプリやモバイルサイトでは3階層くらいの平坦なデータ構造が望ましいとする考え方もあります。

「クリックが少ないほうがよい」というのと似たルールに「一度に7個を超える選択肢を与えてはいけない」というものがあります。人間が7個を超える選択肢を一度に検討することができないという考え方に基づいています。しかしこれも誤りです。



階層が浅い

目的のものに到達するのにクリック1回で済むが、9個の選択肢を一度に検討しなければならない

階層が深い

目的のものに到達するのに2回クリックする必要があるが、その都度3個の中から1個を選べばよい

この「マジックナンバーは7」という説は、1956年に発表されたジョージ・A・ミラーの論文に端を発しています[†]。この論文でミラーは、平均的な人はワーキングメモリ（短期記憶）に7個の項目しか記憶できないと結論づけています。これは正しい説かもしれませんが、選択には当てはまりません。すべての選択肢をワーキングメモリに入れておく必要はないのです。提示された項目を順番に見ていって、その中からもっとも目的に近そうなものを選べばよいからです。この挙動は、1956年に心理学者ハーバート・サイモンによって作られた「満足化」（satisficing）という言葉で呼ばれています。完璧な選択のためにすべての選択肢を比較する必要はなく、ほとんどの人は最初に出会った十分満足のできる項目を選ぶのです。

† Wikipediaにこの論文に関する詳しい解説（英語）があります。http://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

しかし、バリー・シュワルツは“*The Paradox of Choice*”［Sch05］の中で、一部の人は最初の満足できる解ではなく「最高の解」を選ぼうとすると書いています。さらに、ほとんどの人は選択肢が多くても対処はできますが、そうすることを好みはしません。シュワルツは「決断をするのに努力が必要になるので、あまり選択肢が多いと意欲を失わせることにつながる」と書いています。

つまり、ほとんどの人は選択肢が多くても選択はできるが、あまり多すぎるのは好ましいことだとは思っていないのです。

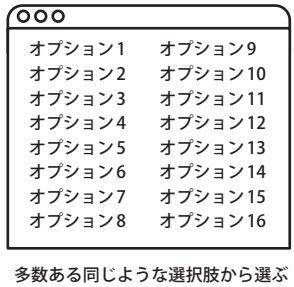
UIとして望ましいのは、できるだけクリック数が少ないものでもなく、また各ス

テップの選択肢をできるだけ少なくすることでもありません。クリックをするときにできるだけ簡単に選択できるようになっていることが大切なのです。ユーザーが明白なゴールをもっているなら、階層の各レベルにおいてゴールに到達できるであろう（あるいはゴールに近づけるであろう）と考える選択肢をユーザーが簡単に見つけられなければなりません。各ステップでゴールに近づいていると感じられさえすれば、深い階層を潜っていくのもさほど気にならないのです。

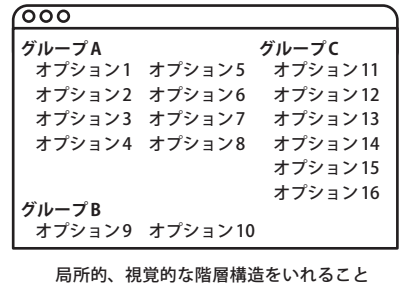
8.7.3 グループ化

ユーザーに、10を超えるような選択肢の中からひとつを選んでもらうのはそれほど難しいことではありません。だからといって、グループ化せずにただ選択肢を並べればよいということではありません。何の構造もなく、単に選択肢が並んでいるものの中からひとつを選ぶのは負荷の大きな情報の提示方法です。

これに対して、選択肢が何らかの意味でグループ化されていたり順番に並んでいたりすれば選ぶのが楽になります。ひとつの画面に表示される選択肢の数は、主に表示するページのデザインによって制約されることになります。階層の特定のレベルで多数の選択肢があるようならば、グループ化しましょう。こうすることで、選択肢に対して「局所的な階層構造」を付加することになり、多数の選択肢から探しているものを見つけるのが楽になります。



多数ある同じような選択肢から選ぶのはユーザーの負担が大きい



局所的、視覚的な階層構造をいれることで選択が容易になる

こうした構造を導入するのには類似したものを近くに置くという方法以外にもさまざまな方法があります。“*Vision Science: Photons to Phenomenology*”［Pal99］でスティーブ・E・パーマーはさまざまな方法を提示しています。

次のような8個の黒丸が並んでいるとします。このように等間隔で並んでいると互いに関係をもっているようには見えません。

しかし、いくつかを少し移動するだけで、8個バラバラではなく、4対の点のように見えるようになります。これが距離（近接性）を用いた構造化です。

次のように丸の色を変えて組にするという方法もあります。

丸の大きさを変えるという方法もあります。

また、楕円形にして向きを変えるという方法もあります。

ここまでで色、大きさ、向きを使ったグループ化の手法を見ましたが、書体を変えるという方法もあるでしょう。

さらには、要素を線で囲んでグループ化するという方法もあります。

また、関係のあるものを線で結ぶという方法も考えられます。

このように構造化やグループ化にはさまざまな方法があります。ひとつの画面でたくさんの要素を表示する際には、こういった手法を利用しましょう。

ポイント

- アプリやウェブサイトに関して、ユーザーが見る画面の階層構造に内部的な構造（たとえば社内の事業部の構成）を直接使うのは避けましょう。
- （何がどこにあるべきといった）ユーザーの考えを知るためにカードソートを使いましょう。
- クリックする回数が最小になるように階層を最適化するのが必ずしも正しいわけではありません。
- 画面に一度に表示される要素を構造化するためにグループ化しましょう。

参考文献

カードソートに関してはドナ・スペンサーの“*Card Sorting: Designing Usable Categories*”［Spe09］がお勧めです。必要なことはすべてこの本に書いてあると言っても過言ではありません。

情報アーキテクチャについて知るには、ピーター・モービルとルイス・ローゼンフェルドの『Web情報アーキテクチャ ── 最適なサイト構築のための論理的アプローチ』［MR06］やドナ・スペンサーの“*A Practical Guide to Information Architecture*”［Spe10］がお勧めです。ドナ・スペンサーはカードソートに関するブログ[†]や情報アーキテクチャに関するブログ[††]を書いています。

人間の知覚について詳しく知りたい方には、“*Vision Science: Photons to Phenomenology*”［Pal99］がお勧めです。

† http://rosenfeldmedia.com/books/cardsorting/

†† http://practical-ia.com

23章 スピード

ユーザビリティテストにおいては、アプリの実行速度や応答速度にあまり注意が払われない傾向があります。あるボタンのラベルの意味が理解しにくいといったことはすぐにわかるのですが、実行速度や応答速度はアプリをある程度使い慣れている人でないと判定できません。テスターと実際のユーザーでは、アプリを使う動機が異なります。テスターはアプリのテストをしているということがわかっているので、アプリに待たされてイライラしても急にテストをやめて他社のアプリに移ったりはしません。

しかし、実際のユーザーは移ってしまうかもしれません。ベンチャーキャピタリストのフレッド・ウィルソンは、これについて次のように説明しています[†]。

> スピード（実行速度）は単なる「機能」ではありません。「もっとも重要な機能」です。アプリが遅ければ、誰も使おうとしません。パワーユーザーより、多数を占める一般ユーザーでその傾向が強いのです。パワーユーザーは、本当に速いウェブアプリを作成する難しさを知っているため思わず同情するような感じで、我慢してくれるのではないでしょうか。しかし、私の妻や子供たちの様子を見ていると（私にとって世界の一般ユーザーの代表です）、何か時間のかかることがあると、すぐにアプリの世界から離れてしまいます。［中略］私たちが投資している会社のアプリの実行速度が落ちると、会社自体の成長も遅くなるようです。スピードは単なる機能ではないということは経験から証明されています。スピードは必須の条件なのです。

† http://thinkvitamin.com/web-apps/fred-wilsons-10-golden-principles-of-successful-web-apps/

ユーザビリティテストで実行や応答の速さが不十分であるという明らかな兆候が見られなくても、速度の問題に対して注意を怠らないことが重要です。

23.1 | 応答速度

まず最初に調べるべきことが、アプリの応答速度です。この問題に関する研究はかなり一致した結果を出しています[††]。処理が終了するまでの時間が0.1秒未満であれば、ユーザーは処理が瞬間的に終わったと感じます。0.1秒以上はかかったものの1秒未満で完了した場合、瞬間的とは感じられませんが、それまでのことがわからなくなったりはしません。1秒を超えてしまうと待たされている間に気が散ってし

†† 詳しくは、ヤコブ・ニールセンの反応時間に関する論文を読んでください。http://www.useit.com/papers/responsetime.html

まう可能性が高くなります。

可能な場合は、操作の結果が常に0.1秒未満で現れるようにするのが最善です。

これと関連するのが、ボタンを指でドラッグして画面を移動させるような操作をナチュラルユーザーインタフェース（NUI）で実現する場合です。NUIについては13章で説明しましたが、ドラッグするときには、動作が連続的に行われるため応答速度が特に重要になります。ユーザーの操作に対するUI側の反応は即時的で滑らかなものでなければなりません。ユーザーの操作に追随できねばならず、しかも高い「フレームレート」で追随する必要があります。応答速度によって、実際の「モノ」を操作していると感じるか、コンピュータに連続的に命令を出しているという印象をもつかが変わってきます。UIを操作中に入力処理やアニメーションの再生などで遅れが発生すると、ユーザーがアプリに対して構築しているメンタルモデルを壊すことになり、大きなストレスの原因となります。

23.2｜進捗状況のフィードバック

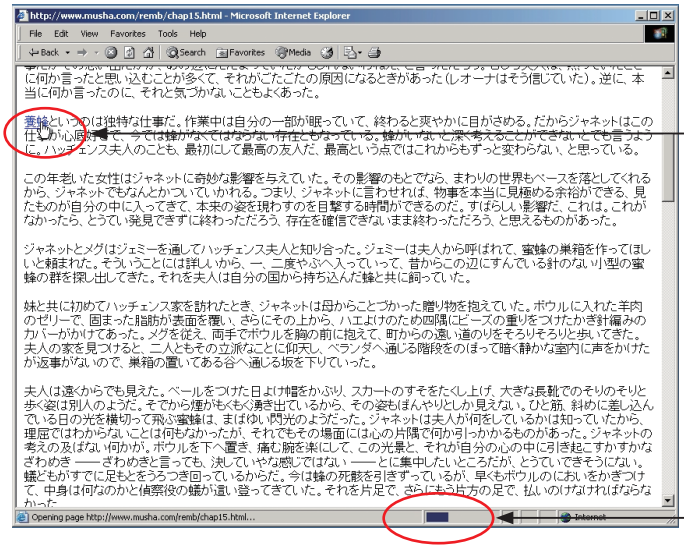
処理が0.1秒で終わらない場合は、何らかのフィードバックをする必要があります。どのタイプのフィードバックが必要かは、処理にかかる時間と処理の種類に依存します。

処理にかかる時間が1、2秒なら、カーソルを砂時計にして、「処理中」を意味するちょっとした表示を付け加えるだけでよく、どこまで処理が進んだかをわざわざ表示する必要はありません。

ロード中…

ここで目標としているのは、処理にどれくらい時間がかかるかを伝えることではなく、ユーザーの命令を受け取って作業中だということをはっきり示すことです。「はっきりと示す」ことがとても重要です。ウェブブラウザを使っている人を観察すると、リンクを繰り返しクリックしている人がいます。最初のクリックがブラウザに認識されなかったと考えているのです。皆さん自身もその経験があるかもしれませんが、なぜこんな行動をしてしまうのでしょうか。多くのブラウザでは、小さく点滅する「ロード中…」のメッセージやプログレスバーが表示されます。そもそもこ

の表示がわかりにくいというのが問題です。さらに、ユーザーがクリックした箇所から離れた場所に表示されるのもよくありません。



ユーザーのクリックをブラウザが受け取ったという明白なフィードバックがないために、何度も何度もクリックしようとする人が現れるのです。

2、3秒より長くかかる処理の場合は、何らかの形で処理がどれくらいかかるかを示す進捗状況のフィードバックが必要になります。プログレスバーを用いるのが普通です。

場合によってはコンピュータが現在どんな作業をしているのかを伝えるのもよいでしょう。伝えることで待たされている理由が明確になる場合は特に重要です。たとえば、動画編集ソフトで動画を編集してウェブにアップロードする場合、作業は通常ふたつの段階に分かれます。第一段階は動画のレンダリングとエンコードです。続く第二段階で動画がアップロードされます。プログレスバーの近くに、現在の作業内容を示す短い説明文か、あるいはちょっとしたアニメーションを追加して、現

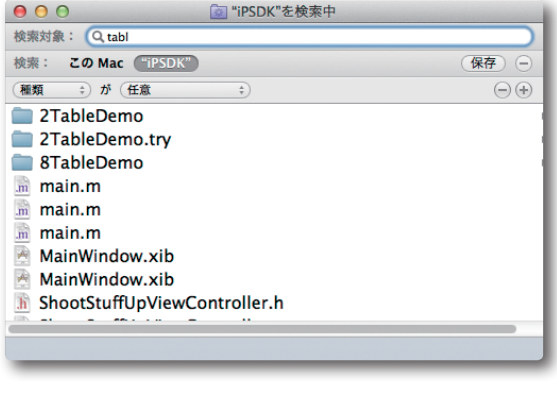
在の作業状況をわかるようにするのがよいでしょう。アプリが現在どのような作業を行っているのかを示すことで、作業全体に時間がかかっている理由をユーザーに伝えることができ、ユーザーのストレスを減らせる可能性が高くなります。

作業に時間を要する場合、ユーザーがほかのことを始めてしまう可能性が高くなります。そのうちに作業中であることをすっかり忘れてしまうかもしれません。作業が終了したことを音でフィードバックするとよいでしょう。そうすればたとえ他のことに集中していても気づいてくれます。

23.3 | スピード感

スピードは体感するものです。ユーザーはアプリの応答時間をストップウォッチで計るわけではありません。実際に問題となるのは、処理に何秒かかるか、何ミリ秒かかるかではなく、ユーザーがどう感じるかなのです。

スピード感を向上させるひとつの方法が、可能なかぎり早期に結果を（部分的であっても）表示し始めることです。検索システムのUIを実装する場合なら、ユーザーに結果を表示するのを検索が完了するまで待ってはなりません。検索結果は見つかったものからすぐに表示し始めましょう。

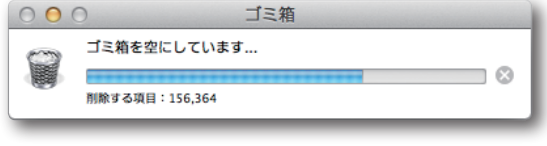


スピード感を向上させる別の方法としては、時間のかかる処理の間もアプリのUIをブロックせず使用可能にしておくという方法があります。作業が終了するまで強制的に待たされるのではなく、その間にほかのことをしていられるのなら、作業に時間がかかっても意識せずに済むかもしれません。

そしてこれが奥の手ですが、プログレスバーに、伸張方向と逆向きに動くさざ波のようなアニメーションを付けると速く動いているように見えます[†]。

† 詳しくは次のページを参照。http://www.newscientist.com/blogs/nstv/2010/12/best-videos-of-2010-progress-bar-illusion.html

プログレスバーは右に向かって伸びるので…



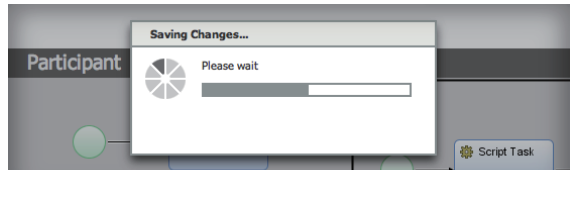
プログレスバーのさざ波模様は左に向かって動くようにする

23.4 | 速度制限

遅いアプリは困りものですが、逆もまた真です。処理が速すぎることもときにはあります。典型的な例はスクロールです。アプリのスクロールが速すぎると、目的の場所を通り過ぎてしまいます。アプリの実行速度を意図的に遅くして、ユーザーが追いつけるようにすることが求められることもあります。

筆者がAppwayというビジネスプロセス管理ソフトの仕事をしていたときの話です。プロセスエディタには［保存］機能があって、変更したときにユーザーが手動で保存できるようになっていました。ところが、ユーザーは保存用のボタンを何度もクリックしてしまいます。時には、「きちんと動いていない」と苦情を言う人もありました。

調査の結果、保存が「速すぎた」ということがわかりました。ワークフローの保存には1秒もかかっていませんでした。ドキュメントが実際に保存されことを示すフィードバックがなかったのです。ユーザーにとってはドキュメントの保存は非常に大切なことですから、ボタンをクリックしても何も起こらないのがとても不安に感じられたのです。解決策として、0.5秒ほどで消える「ニセの」プログレスバーを表示しました。



こうすることで、何か実体のある作業が行われていることをユーザーに伝えました。UXコンサルタントのハリー・ブリグナルは、硬貨計数機について同様の話を紹介しています[†]。その機械は硬貨の計数速度が非常に速かったので、顧客は正しく計数できていると信用しなかったようなのです。メーカーは結果を遅く表示するように「UIを改良」し、カウンターが最終結果を表示するまで待つ間に硬貨を計数する効果音を流すスピーカーを追加しました。これにより「UXは改善」されました。ユーザーは機械が自分のお金を相応の努力をして数えていると感じるようになり、結果を信じるようになったのです。

† http://www.90percentofeverything.com/2010/12/16/adding-delays-to-increase-perceived-value-does-it-work/

ポイント

- アプリにとって反応速度とスピード感は何にも増して重要な性能です。ユーザビリティテストでは、この種の問題は明らかにならないのが普通です。
- 瞬間的に終了したと感じてもらうには、処理が0.1秒未満で完了しなければなりません。
- 処理に1秒以上かかる場合は、アプリが作業中であることを示す何らかの標識を表示しなければなりません。この標識は、ユーザーが視線を向けている場所に表示します。
- 知覚される速度が実際のスピードより重要です。時にはより速く感じさせるための演出が有効です。
- 処理が速すぎることもありえます。ユーザーエクスペリエンスを向上させるためには、処理に時間がかかっていると感じさせる演出が有効な場合もあります。

参考文献

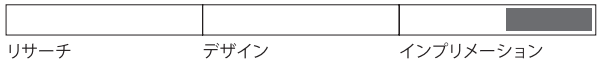
ヤコブ・ニールセンの論文は、応答時間に関する貴重な情報源です[††]。

ブルース・トグナッツィーニは、論文 *"First Principles of Interaction Design"* の中で遅延時間について解説しています[‡]。

†† http://www.useit.com/papers/responsetime.html

‡ http://www.asktog.com/basics/firstPrinciples.html

33章
A/Bテスト



概要

ここまでは、主に個々のユーザーがどう行動するかを観察するユーザビリティテストについて説明してきました。これは製品のデザインに伴う問題点を見つけるためには良いのですが、2種類（あるいは3種類以上）の異なったデザインを比較して最良のものを見い出す必要がある場合には適していません。

たとえば、無料会員登録ページの文言を決めようとしていて、訪問者をその気にさせるにはどれがよいか確信がない場合です。あるいは、ログインフォームの配置を2パターン用意したものの、ユーザーにとってどちらか便利かよくわからない場合などです。そうした疑問に答えるためのもっとも一般的な方法が、A/Bテスト（A/B Testing）です。A/Bテストを実施すれば、ふたつのうちのどちらのデザインが良いかを見つけ出すことができます。

「A/Bテスト」と呼ばれてはいますが、必ずしも異なるデザインふたつだけを比較しなければならないわけではありません。もっと多くの異なるデザインを比較の対象とすることもできます。この理由で、時には「A/B/nテスト」と呼ばれることもあります。

なぜ役に立つのか

A/Bテストを実施すれば、確かなデータに基づいてデザインを改良できます。当て推量は入りません。複数の選択肢の中から最良のデザインを選ぶときには、A/Bテストが有効です。

前提条件

動作する製品があり、それを相当数の人に利用してもらう必要があります。

33.1 | いつA/Bテストを行うか

A/Bテストはふたつ以上のデザインを比較してどれが良いかを見つけ出すのに有効です。一般的に言ってこれが役に立つ状況には、次のふたつがあります。

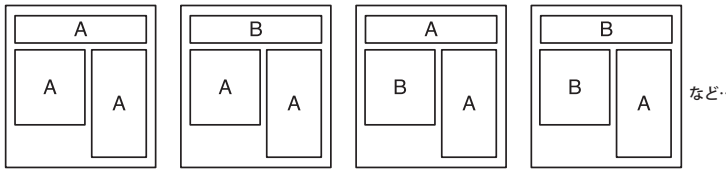
- ある製品をデザインし直したので、新しいデザインが以前のものよりも良くなったかを知りたい。

多変量テスト

多変量テスト（Multivariable Testing）とは、A/Bテストの一種です。正規のA/Bテストの場合は、完全に実装されたデザインをテストします。

A/Bテストでは完全に作り上げられたデザイン同士を比較する

多変量テストでは、製品の異なる部分を一度にテストします（製品のデザインの中に複数の変量があるので、このことから多変量テストと呼ばれます）。



多変量テストでは、ひとつのデザイン内のいくつかの部分について比較する

多変量テストは、あるデザインのうちの小さな部分に関するA/Bテストを多数同時に実行するものであると考えてください。多変量テストは、それを行うしっかりした理由がない場合、調和を欠いた要素を組み合わせたフランケンシュタインのようなデザインになりかねないので、筆者は実施しません。多変量テストで最高点を取る個々のデザイン要素を10個組み合わせても、すばらしいデザインにはなりません。通常はそれよりも、一貫性をもって完全に作り上げた2種類以上のデザインを相互に比較テストして、いずれが優れているかを調べたほうが良い結果が得られます。

- ある問題に対していくつかの解決法が考えられ、どれがもっともうまくいくのかを見極めたい。

A/Bテストは、レイアウトの大幅な変更から色や言葉づかいの小さな変更に至るまで、あらゆることの比較に使えます。

GoogleやAmazonと同じように毎日数百万の人が皆さんのサイトを利用しているのなら、ユーザビリティのほんの小さな差異で数万人が問題に遭遇するかしないかの違いが生じかねません。そういったサイトやアプリはあまり多くはないでしょうから、皆さんの製品について、ユーザビリティの小さな差異が大きな問題になる可能性はあまりないでしょう。したがって、小さな変更をすべてA/Bテストにかける必要はありません。しかし、大規模な変更や、その製品の特に重要な領域（たとえば、ウェブサービスの会員登録画面）に関する変更を行った際には、A/Bテストを行って有効性を確認しましょう。

33.2｜成功とは何か？

A/Bテストの基本的な考え方は、異なったふたつのデザインを実装してどちらのほうがよりよく機能するかを確認することです。しかし、これには本質的な問題があります。「よりよく機能する」とは具体的にはどういうことでしょうか？ ユーザーが皆さんの製品を使って「成功した」と言えるのはどの時点でしょうか？ その答えが、明らかな場合もあります。たとえば、精算システムのデザインなら、ユーザーが精算を完了した時点でシステムが「機能した」と言えます。しかし、何をもって「成功」と言えるのかが直ちに明らかではないことも多いのです。

「成功」を定義するひとつの方法は、デザイン過程のごく初期に立ち返ってユーザーの目標について考えることです。その製品を使う理由がわかれば、成功を定義できます。つまり、目標を達成できるユーザーの比率が高いほど、その製品はよりよく機能しているということです。

時には、ウェブサイトのリンクをクリックする人の割合やウェブサイト訪問者のうちで会員登録した人の割合を求めるだけでよい場合もあります。そうでない場合には、この質問に答えるのはもっと難しくなります。作業を開始した人のうちで実際に完了したのは何人でしょうか？ 成功の定義は製品によって変わります。

33.3 | テストの準備

ここまでくると、手元にはテストしたいデザインがふたつ（あるいはそれ以上）あり、成功の定義もできています。次は、両方のデザインを実装して、それぞれを別々のユーザーに分けて提供する手段を見つけなければなりません。

製品がデスクトップアプリの場合は、ふたつ以上のビルドを作成して、それぞれを別のユーザーに配布するか、両方のデザインを同一のアプリに組み込んで、初回起動時にいずれかのデザインを選択するといった方法をとる必要があります。

製品がウェブサイトの場合は、ふたつのデザインに異なるURLを割り当てるか、ユーザーが単一のURLにアクセスしたときにどちらのデザインを表示するかを決めればよいでしょう。

ウェブサイトのA/Bテスト用に多数のオンラインツールが提供されていますが、Google Analyticsはその代表格でしょう[†]。Vertster[††]もそうしたツールで、ウェブサイトに対する多変量テストを行うのに有用です。

33.4 | テストの実施

A/Bテストを実施する際には、個々のユーザーが使うデザインがふたつの間で切り替わらないようにする必要があります。サイトを再訪したり、アプリを再起動したりするたびに、別のデザインが現れては混乱してしまいます。

テスト結果を収集する方法も必要です。ウェブサイトの場合は、（各ユーザーが何をしているかが「見える」ので）簡単ですが、デスクトップアプリの場合には簡単ではないかもしれません。

また、皆さんが利用データを受け取っていることをユーザーに知らせ、どんなデータを受け取っているのか、なぜそれが必要かを具体的に説明する必要もあります。Googleは、Chromeのダウンロードページでこれを行っています。

† Google Analyticsを使ってウェブサイトのA/Bテストを実施する方法は、ジョージ・パーマーのエッセー（http://rowtheboat.com/archives/39）を参照してください。Googleはウェブサイトの変更をテストするために、Website Optimizerを 提供していましたが、2013年1月 現 在、Google Analytics Content Experimentsを使うよう推奨されています。

†† http://www.vertster.com

33.5 | テスト結果の解釈

ほとんどのA/Bテストツールは結果の解釈もしてくれます。これを見れば多くの場合どのデザインがもっとも機能するかその傾向を素早く確認できます。しかし気をつけてください。人間はパターンを求める傾向があり、何もないところにパターンを見つけてしまいがちなのです。統計的に見て、目の前にある結果が偶然得られる確率がどれだけあるかを把握する必要があります。

User Effectのウェブサイト[†]には、ふたつのデザインに対するA/Bテストの結果が統計的に有意であるかどうか示してくれるページ[††]があります。有意でない場合には、有意な結果を得るためにあと何人の訪問者が必要かも教えてくれます。

3種類以上のデザインでA/Bテストを行う場合には、個々のデザインを比較すれば、差異が重要なものかどうかを調べられます。

† http://www.usereffect.com

†† http://www.usereffect.com/split-test-calculator

ユーザーの区分け

デスクトップアプリをテストする場合に、ユーザーをふたつのグループに分けるのは簡単です。ユーザーがアプリをダウンロードする際に、ランダムにどちらかのビルドを選びます。通常、各ユーザーが同じバージョンをダウンロードするのは一度だけですから、別のデザインを目にしてしまうユーザーのことを心配する必要はありません。両方のデザインを同一のビルドに含めた場合には、初回起動時にランダムにどちらかのデザインを選択するようにし、それを記憶しておきます。

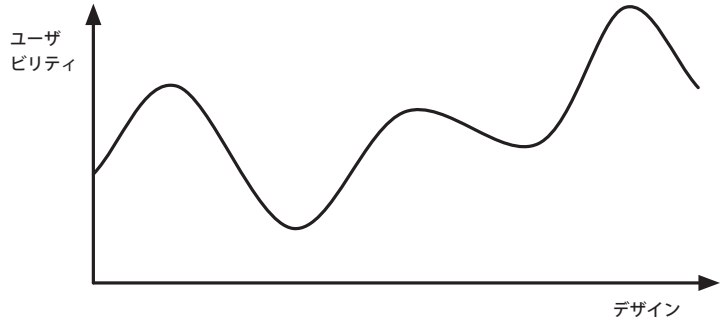
ウェブサイトの場合には少し複雑になります。たとえば、訪問者のIPアドレスからハッシュ値を計算し、その結果の剰余をとって決める方法が考えられます[†]。どちらのデザインを採用するか決めたら、クッキーに記憶しておきます。再訪問した場合は、クッキーの値に従って、どちらかを選びます。ユーザーがクッキーを許可していない場合やIPアドレスが変わってしまった場合は、別のデザインが選ばれてしまう可能性がありますが、こういったユーザーの数はごく少数でしょう。

† IPアドレスはランダムではありませんから、単にIPアドレスをもとに剰余をとるだけでは各区分のユーザー数に偏りが生じます。ハッシュ関数で値を変換してから剰余をとればこの偏りが除かれます。

33.6 | 留意点

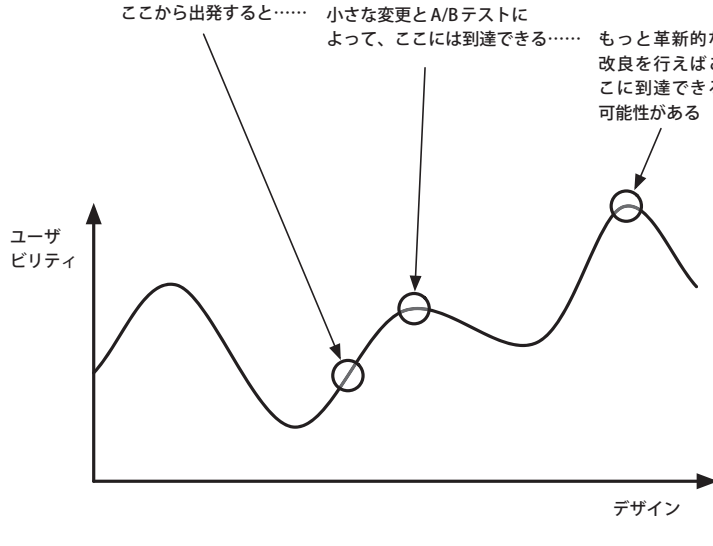
小規模なデザインの変更を繰り返す場合、A/Bテストだけではユーザビリティを局所的に最大にするだけであることに注意が必要です。変更を加えていくことで、良くはなっていきますが、それによって最高のものにたどりつけるとは限らないのです。

実現可能なすべてのデザインを考えてみてください。そのうちのいくつかは使いやすく、他のものは劣ります。それをグラフにプロットすると、次のようなものになるでしょう。



デザインを変更すると、ユーザビリティが変化します。現時点でデザインの質がこのグラフの中程にあるならば、小さな変更とA/Bテストを繰り返すだけでは最高のデザインには到達できないでしょう。こう考えてください。山登りに行き、常に上へ上へと登ることに決めたら、最終的にその山の頂上には到達してもその隣の山の頂上には決してたどり着くことできず、このとき隣の山のほうが高い場合もありえます。その高い山の頂上に行くには、まずは下らなければなりません。

同様に、最終的にはるかに優れたデザインに到達するには、劣ったデザインからやり直さなければならない場合もときにはあります。



こうした問題を避けるためには、デザインの「累積的な変更」だけでなく、「大胆な変更」も検討する必要があります。A/Bテストはすべてのデザイナーにとって優れた手段ですが、限界もあることに留意してください。

ポイント

- A/Bテストを行うと、異なったデザインを比較してどれがもっともうまく機能するかを見い出すことができます。
- 「機能する」ことの定義は、ユーザーの目標によって変わります。
- 利用データを収集するのは、ウェブサイトの場合は容易ですが、デスクトップアプリやモバイルアプリの場合はもっと難しくなります。ユーザーの了承を得ることと、利用目的を具体的に告げることを忘れないようにしましょう。
- 得られたデータに基づいて決定を下す前に、データが統計的に有意なものかを必ず確かめましょう。
- 高度に最適化した旧デザインを新しいデザインと比較するときには、直接的な比較で新しいデザインが劣っていたとしても、旧デザインにしたのと同様に磨きをかければ最終的にそれを凌駕できる場合があることを心に留めておいてください。