知能情報実験3：データマイニング班

(week 1) 開発スタイルと実験で使う環境

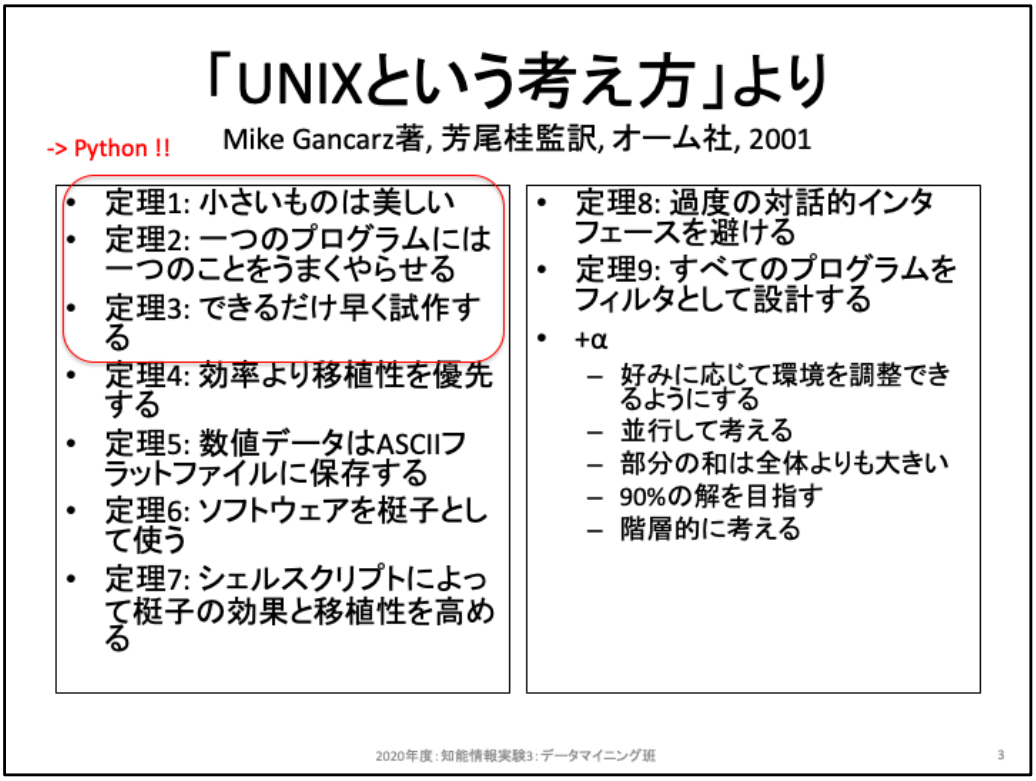
1. UNIX哲学
2. 実験で使う環境
3. （アジャイル）

実験ページ: http://ie.u-ryukyu.ac.jp/~tnal/2020/info3/dm/

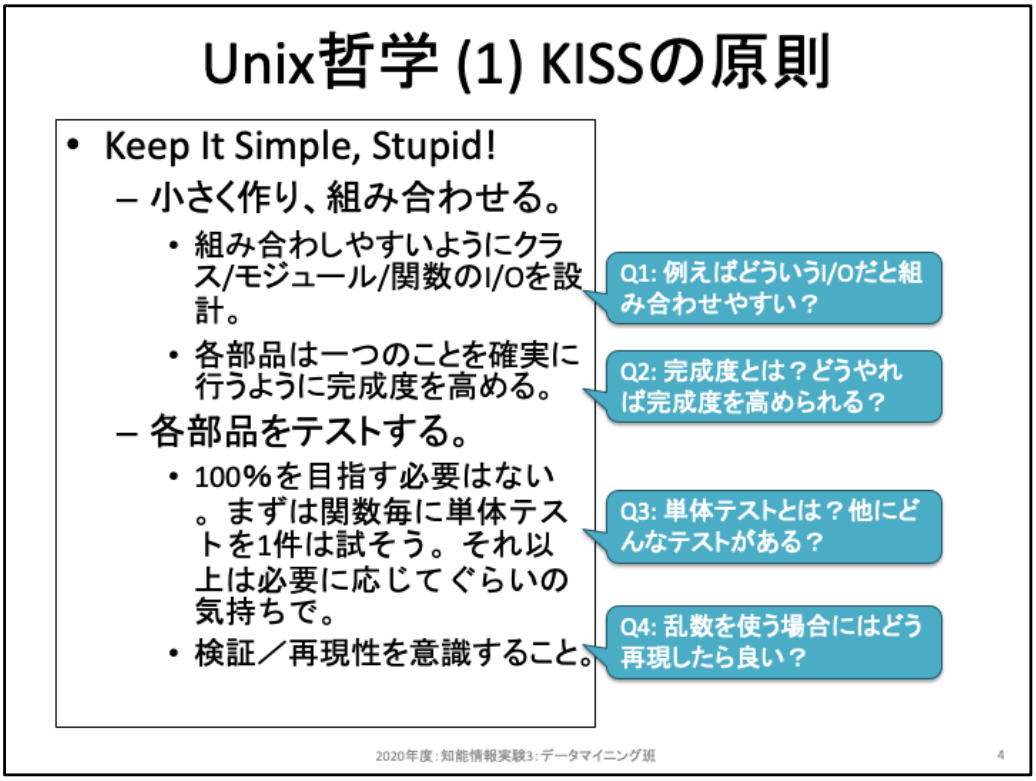
知能情報実験3：データマイニング班

(week 1) 開発スタイルと実験で使う環境

1. UNIX哲学
2. 実験で使う環境
3. （アジャイル）



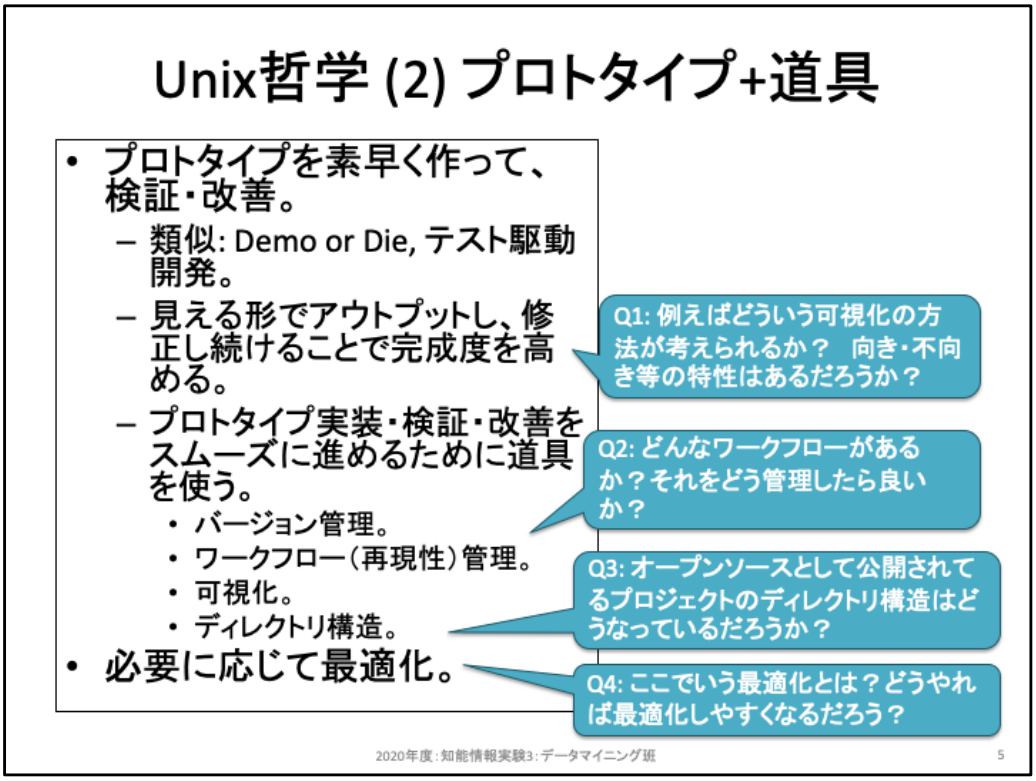
Unix系OSはPOSIX標準化を経て再配布自由・改変自由なUNIXクローンの結実に繋がり、広まった。様々な特徴を有するが、その開発指針やそもそもの考え方は様々な開発現場において有用である。ここではその冒頭に出てくる3つの定理が示す意図を汲み取ってみよう。



定理1のKISS原則は、何を作るにしても小さく作り、組み合わせろというものだ。個々の関数・クラス・モジュールを小さく設計し、それらを組み合わせることで目標を達成することを指針とする。その逆に、数百行～数千行からなる1つの関数を作ってしまうと、その関数に不備があった際にはどこが問題なのかを発見することが難しく、また、新しく機能を追加したい場合にも様々な部分が影響し合うために改変（バージョンアップ）しにくい。これに対し、一つ一つの関数が正しく動作していることを確認しやすい粒度で設計できれば、書き上げたコードが想定通りに動作しているかどうかを検証しやすく、問題があった場合にはその箇所を特定しやすいため、修正しやすい。このため、KISS原則を頭においた設計が重要視されている。

さて、KISS原則の考え方をもう一歩踏み込んで深い理解につなげてみよう。

今、ある機能をコードとして書き上げたい状況だとしよう。その機能は2個の関数に分けて実装し、組み合わせることで最終目標を達成する。このとき、関数を組み合わせしやすくするためにはどういうI/O（Input/Output）にすると良いだろうか（スライドQ1）。また、各関数の完成度はどのように計測したら良いだろうか。そもそも完成度とは何だろうか（Q2）。完成度に関連して、単体テストは何だろうか。それ以外にどのようなテストがあるだろうか（Q3）。関数で乱数を利用した機能がある場合、どのようにテストするとよいのだろうか。また乱数を再現することはできないのだろうか（Q4）。



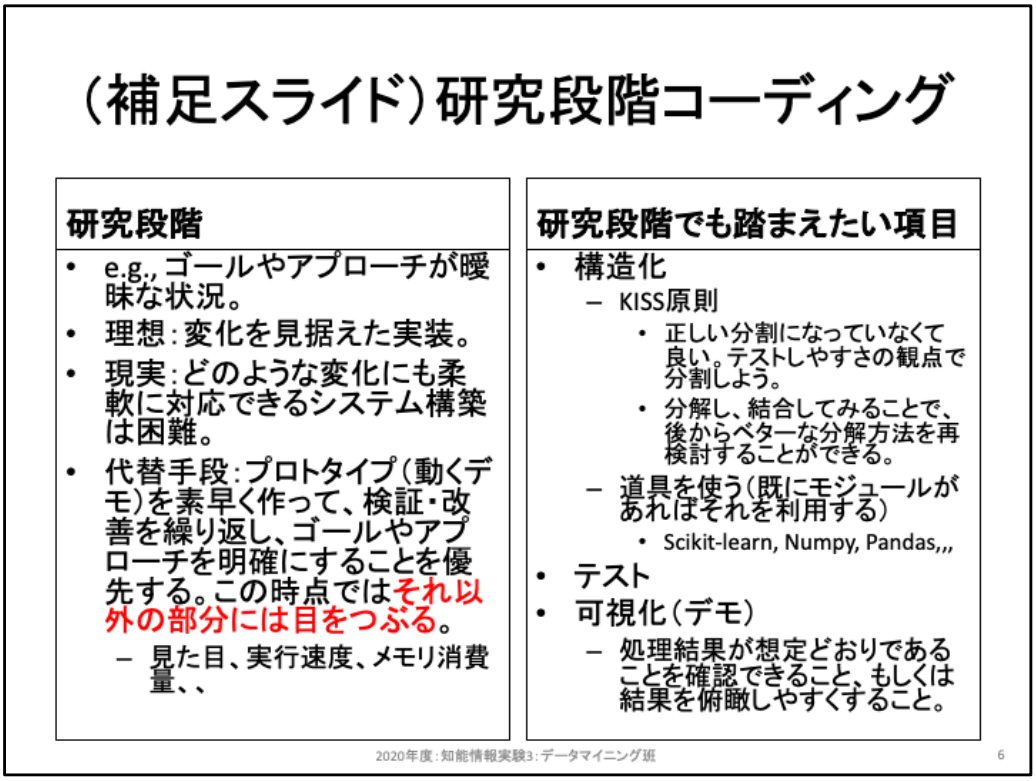
KISS原則と似通っている部分もあるが、第2の哲学「プロトタイプを素早く作る」を眺めていこう。

開発は、(1)仕様が厳密に定められている状況と、(2)仕様が曖昧で変更発生がありうる状況に大別される。(1)は、予め仕様策定の時点で厳密なやり取りを通してやること・やらないこと・達成したいこと・どう達成するか・その計画等、あらゆることを開発フェーズに入る前に決め、それから開発に取り組むことが多い。このようなアプローチを「ウォーターフォール型開発」と呼ぶことが多い。なぜウォーターフォールと呼ぶのかは調べてみよう。

ウォーターフォールは、一度仕様を決めたあとで変更が一切無いのであれば構わないかもしれないが、これが向いていない状況は多々ある。例えば、開発を依頼しようとしている顧客がやりたいことを明確にリストアップできているとしても、その仕様に沿ったシステムが最適である保証はない。発注者の仕様はあくまでも「その部署が考えた最高な仕様」に過ぎず、実際にはそのシステムを利用する全社員にとっては使いづらいかもしれないし、他のシステムとの連携が取れておらず逆に手間が増えてしまうかもしれない。また、実験や研究においては結果を確認しながら随時アプローチを変更していくことが多い。これらを一言でまとめると「そもそもそれまでに存在していないシステムの最適解(最適な仕様)は誰にも分からない」となる。このような状況で、仕様を固めてから開発に取り組むという作業方法は非効率的である。

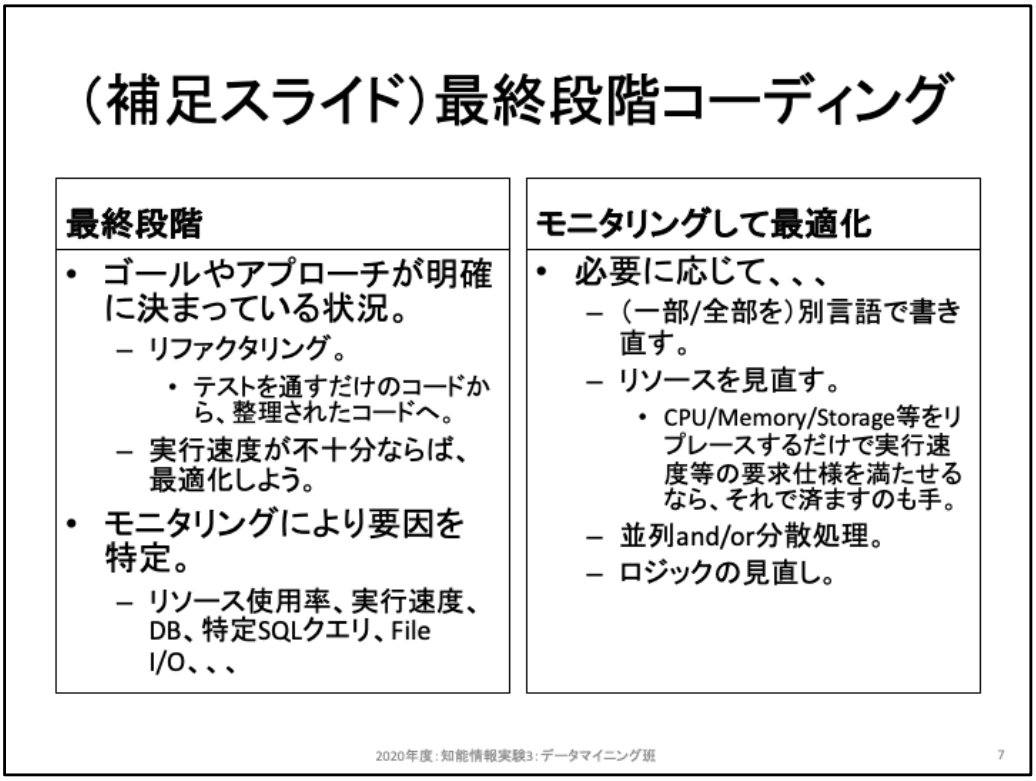
このような背景を踏まえた考え方が「プロトタイプ」や「Demo or Die」である。明確な仕様が存在していない状況で、分かる範囲の小さな仕様からそれを実現する方法を模索し、実際に作り上げてみる。そのアウトプットを顧客とともに確認しながらよりベターな解決策を模索していく。アウトプットが無いと確認することが困難であるため、Demoしろ(プロトタイプを作れ)。そのようなニュアンスがDemo or Dieという標語に込められている。

アウトプットとして見える形に出すためには、どのような方法があるだろうか？(Q1)　また、それぞれの向き・不向きはあるだろうか？　ある程度プログラムが複雑になってくると、関数やファイルが多数存在する状況になる。このような状況でアウトプットを再確認しやすくする(ワークフローを明瞭にする)にはどのような方法があるだろうか？(Q2)　第三者がわかりやすいディレクトリ構造はどうあるべきだろうか？(Q3)　開発途中に重要視すべきことと、そうではなく優先度を落とすべきことにはどのようなことがあるだろうか？最適化とは何だろうか？(Q4)



前スライドの「仕様が明確に決まっていない状態」の例として、研究段階の開発についてまとめたもの。より詳細は下記を参照してみよう。

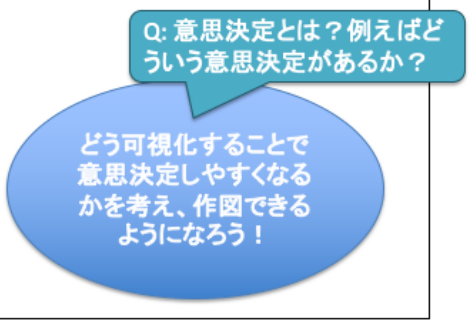
研究者流コーディングの極意 by 東北大学・岡崎先生
http://www.chokkan.org/publication/coding-for-researchers.pdf



「研究段階」においては素早くデモを繰り返すことを最優先する。これらを通しある程度見通しが立ち、仕様確定に近づいてきたら、適宜リファクタリングしよう。リファクタリングとは機能の修正や追加なしに、コードを見やすくするためやリソースの無駄を省いたコードに改善すること等を目的としてコードを書き直すことである。機能が変わってはならないため、コード修正前後で動作が変わらないことをテストにより確認しよう。

（補足スライド）可視化

- 数値化／テキスト化／グラフ化
- 表
  - 表計算ソフト, HTML
- 線グラフ、円グラフ, 棒グラフ, 帯グラフ,散布図, 箱ひげ図
- 2次元、3次元軸上のグラフ
  - gnuplot, matplotlib, google charts
- データ構造やグラフ理論の「ノードとエッジのあるグラフ」
  - Graphviz (dot), D3.js, draw.io
- ヒストグラム
  - matplotlib, Google charts
- レーダーチャート
  - Google charts
- デモ
  - HTML, JavaScript, CSS
- コード／インタラクティブ
  - IPython Notebook -> Jupyter



知能情報実験3：データマイニング班
(week 1) 開発スタイルと実験で使う環境

1. UNIX哲学
2. **実験で使う環境**
3. （アジャイル）

実験で使う環境(推奨/目安)

- プログラミング言語
  - Python 3.6以上
- 補足:
  - Mac OS X 10.15 標準では Python 2.x系列、Python 3.x系列両方がインストール済み（多分）。
  - Python 2.x系と3.x系は互換性が無い。
  - Python 2系は開発終了(->少し延長)。既に2系への対応をしていないライブラリも出始めている。
  - Python 3.x系は、全ての文字列がデフォルトでUnicodeになり、文字列処理がしやすくなった。
- 参考: Python 2 と Python 3 のどちらを使って開発すべき？
  - http://nyagao.hateblo.jp/entry/2014/03/25/210415

- 開発環境
  - PyCharm or Editors(Emacs, Vim,,,)
  - VS Code (Visual Studio Code)
- バージョン管理
  - 推奨: Git + GitHub
- Python仮想環境
  - 推奨
    - miniconda + conda
- 機械学習パッケージ
  - Scikit-learn
    - Python (NumPy, SciPy, matplotlib)
    - 上記が動作するなら環境構築方法は自由。
- （自然言語処理）＊オマケ
  - Mecab, NLTK
  - 単体テスト
    - pytest, **doctest**

知能情報実験3：データマイニング班
(week 1) 開発スタイルと実験で使う環境

1. UNIX哲学
2. 実験で使う環境
3. （アジャイル）

アジャイルソフトウェア開発宣言

http://agilemanifesto.org/iso/ja/

- 前提
  - 事務的な作業と異なり、新しいモノを生み出す作業の場合、その「新しいモノ」は仕様を厳密に確定することは困難。
  - ***必ず***何かしらの仕様変更が入る。
- アジャイルソフトウェア開発宣言
  - 左記のことがらに価値があることを認めながらも、私たちは右記のことがらにより価値をおく。
    - プロセスやツールよりも個人と対話を、
    - 包括的なドキュメントよりも動くソフトウェアを、
    - 契約交渉よりも顧客との協調を、
    - 計画に従うことよりも変化への対応を、
  - 「動くソフトウェア、変化への対応」のために
    - UNIX哲学（KISS＋プロトタイプ＋道具）
    - YAGNIの原則（実際に必要になるまでは機能を追加しない）