Experiences in taking research code into production

Albert Visagie

Stellenbosch University, Electrical and Electronic Engineering, DSP Group.

CatchWord Language and Speech Technologies.

18 April 2007

Outline

- Problem domain.
- Properties of software and development effort.
- Overview of the architecture.
- Some of the things we did right and some mistakes.
- A short list of best practices.

Speech Recognition 101

- Datasets: *Speech recordings, linguistic annotation.*
- Signal processing: *features, enhancement.*
- Models: *Endless variations of Hidden Markov Models (HMMs), GMM, SVM, CART, Neural Networks.*
- Algorithms: *Training of models, application of models: speech decoding, speaker verification etc.*

User Community

- Students, researchers and me.
- Most of the “developers'” mandate is to do research projects and publish papers, not develop software.
- They are inexperienced.
- They are also the primary users.
- The Professor.
- I want to sell things that contain the software, and
- I have to meet deadlines and fix things.

Historical Perspective

- Started in 1992.
- Students were building things, and their code died when they left.
- Dire need for some reuse.
- C++, no exceptions, no templates.

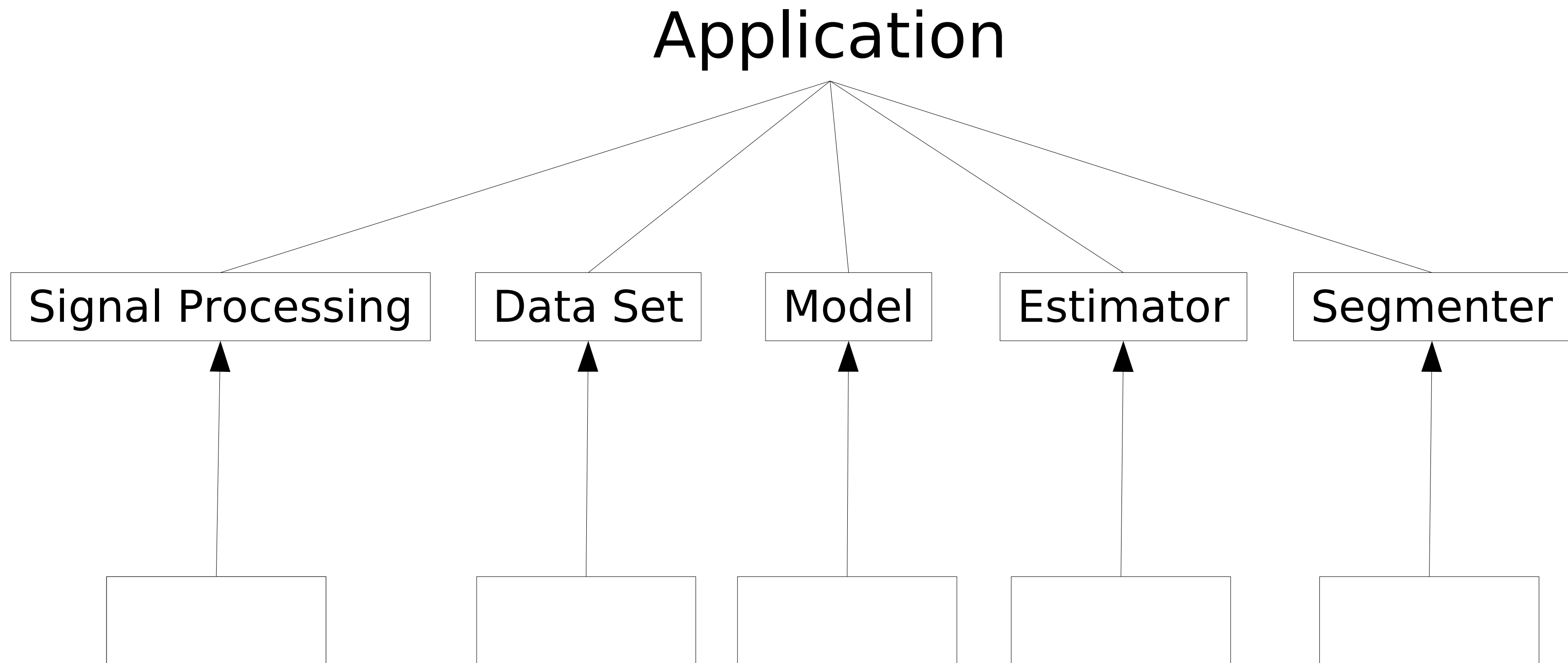
Properties

- Clever things in the code.
- 2 years, masters thesis to create 2 - 5 components.
- Focus on flexibility over packaging and documentation.
- Since the authors are the primary users, their motivation to take short cuts on process & tests is quite high.
- Many different authors, 15 years. 130000 non-trivial LOC.
- Mostly inexperienced coders.
- Linux & extensive bash.

Production

- SpeechGrep: A search engine for speech.
  - Running at three sites (Windows, Java/JNI).
  - Participated in the NIST Spoken Term Detection (STD) 2006
- Speech Detector for broadcast radio monitoring.
  - 3% false accept, <1% false reject.
  - First prototype took ½ day.

Architecture: Components



Architecture: Components

- All models & algorithms are encapsulated as components of various types ➡Abstract classes.
- The base classes store exemplar instances of all implementations.
- Some boilerplate code.
  A typical student project only creates a few components over a year – copy-and-paste from a component template seems fine.
- Greater encapsulation of components. *Mindset.*

Architecture: Workflow

- This gives a very flexible system with interactive configuration of experiments.
- Each component is typically a tree of objects
- Each object has to configure itself and call its constituent objects to configure themselves . . .
- Good news: this architecture lets you build things from scratch,
- Bad news: You have to.

Good Ideas

Good Ideas: Separation of Concerns

- Algorithms, Models, Data-sets.
- Kept data and experiments separate from the core library.
- Components have served well for reuse $\Rightarrow$ minimising interference.

Good Ideas: Components

- Minimised dependencies & spaghetti – Maximised reuse.
- Kept data and experiments separate from the core library.
- Serialization.
- A natural way to package and ship models.
- A great way to separate student code from mature code.
- Patterns for implementing composition & construction.

Good Ideas: C++

- 12 years ago – it really was the best thing around.
- Efficiency – speed and memory.
- Used a sensible subset of the language. Mostly.
- Stuck to the standard. Mostly.
- Slow evolution of the standard and library.
- Simple lifetime control and memory management conventions – standardised, patterns.
- Portable.

Good Ideas: Reinvent the Wheel

- Very little third-party code in broad use.
- Where it is needed, hide it behind custom local interfaces.
- Maintenance hassles...
- Example: home-brew linear algebra classes.

Mistakes

- Mostly things we didn't do, out of ignorance.

Mistakes: Error Handling

Before:

- abort(); CAUTION();
- Inconsistent.
- Now run that through JNI . . .

Solution:

- Simple! Retain benefits of old “system”.
- ASSERT(C, M), and CHECK(C, E, M). That is all.
- Lots of talking and coaching.

Mistakes: Instrumentation

- Situation: “Help, it died.”
- Trace logging!
- Simple conventions, only 5 levels, clear ways of making the decision.
- cout/cerr, commented out.
- Macros.

Mistakes: User Interface

- Previously: Model how we think about signal processing, statistical modelling, pattern recognition.
- Now: Must model how we think about making speech recognisers.
- Solution pending . . .

Mistakes: DRY

- Examples: configuration, serialization...
- Snippets of code to format the serialization of vectors everywhere.
- Current test harness.

Mistakes: JIC instead of JIT

- Examples: classes for dealing with lexicons & annotations.

Lessons Learned

- Choose the language and platform for longevity, available resources & material.
- Minimise dependencies
- Get someone who has done software to put the base and the culture in place.
- People will do things in the easiest way, with the tools nearest at hand. Make sure it is the correct things.
- Light-weight process and conventions.
- JIT, not JIC.

Best Practices: Code

- Modularity: no side effects, simple interfaces, no globals, refactorability, documentation at least on the component level, DRY.
- Robustness: use ASSERT and CHECK royally. Exceptions give application developers a handle on the code.
- Trace logging: Instead of cout/cerr and then commenting them out.

Best Practices: Process

- Version control:
  - Work in trunk,
  - Branch for stabilisation or large changes.
- Code review: via SVN commit emails with diffs.
- Testing:
  - unit-tests, light-weight, run always, no external data dependencies.
  - Once they've tasted the fruit...

Conclusion

- Small team, with extremes:
  - Big difference in skill & experience.
  - Consistent yearly 30% churn.
- The most important thing to get right is the culture:
  - Patterns,
  - Conventions,
  - Consistency, and
  - Communication.