

Software Construction

Prof.Dr. Bruce W. Watson
b.w.watson@tue.nl

2005
(last presentation of this course)

Preliminaries

Aim: To give you a feeling for issues in modern software construction. This includes: *components, toolkits, generic programming, meta programming, patterns (idiom, design, or architecture)* and *domain-specific languages, aspect-oriented programming, . . .*

Presentation style:

- Interactive: don't wait until you're lost.
- Questions can be in Dutch, English, Afrikaans, . . . (perhaps in a Canadian/South African accent).
- This course is somewhat modularized.

Timing:

- 13h30–15h30 on Wednesdays and Thursdays.
- This year we have a guest lecture by Tamas Kozsik, on Aspect-Oriented Programming.
- No lectures in certain periods.

Getting a mark:

- Project: will be announced in the second-half of the course.
- As a preview (one possibility):
 - Done in groups of 2–3.
 - You will pick a field, taxonomize the solutions in the field, design and implement a toolkit, and finally either collect performance data or build a small domain-specific language on top of the toolkit.
 - Will conclude with a report.

Some books to start with:

- “Design Patterns” , by the Gang of Four, Addison-Wesley, 1995.
- “Generic Programming and the STL” , by Austern, Addison-Wesley, 2000.
- “Designing Components with the C++ STL” , by Buschmann, Addison-Wesley, 1998.
- “Modern C++ Design” by Alexandrescu, Addison-Wesley, 2001.
- “Generative Programming” by Czarnecki and Eisenecker, Addison-Wesley, 1999.
- “Pattern-Oriented Software Architecture: A System of Patterns” , by F. Buschmann et al, Wiley 1996.

- “Software Architecture: Perspectives on an Emerging Discipline”, by M. Shaw and D. Garlan, Prentice Hall 1996.
- “Component Software: Beyond Object-Oriented Programming” by Clemens Szyperski, Addison-Wesley, 1998.

Caveats:

- Very new field.
- Still being sorted out; some wishy-washy aspects to the field.
- I assume some competence in C/C++/Java. You may also choose to do your project in another language.

Motivation

Observation: All fields of engineering eventually progress to componentization.

How do things mature in engineering in general.

- Art: Things are done with experimentation and intuition but no externalization.
- Craft: Best practice is passed from master to apprentice.
- Discipline: Someone codifies the best practice in a set of rules, guidelines and principles; enforcement/encouragement structure is provided.
- Science: The codification is formalized and a model is developed to support each of the discipline elements.
- Engineering: Abstraction is raised and work is done in terms of components — at the level of pre-built (and already understood) designs.

What's changing in each step?

Engineering example: Cars.

- General resemblance amongst all cars.
- Reasons: Public acceptance, engineering discoveries, partially-solved designs make things easier.
- Some variants have been tried: 3 wheelers.
- Binding times: when must choices be made? (implemented in, e.g., Mercedes-Benz's configuration language).

Engineering example: Aircraft.

- General resemblance amongst all airplanes.
- Reasons: public acceptance, engineering discoveries, partially-solved designs make things easier.
- Some variants have been tried: forward sweep (why? problems?), engines above wings (Fokker; why? problems?).
- Binding times.

What do we need to make this a software reality?

- Patterns.
- Components.

Later, we see that these two concepts are closely connected.

Components are often codifications of patterns.

Patterns give tried and true ways of arranging and connecting components.

Patterns: concept borrowed from architecture (Alexander).

“[A pattern is] both a description of a thing which is alive, and a description of the process which will generate that thing.”

Experts typically do not solve from scratch — reuse previous designs.

Three things in describing patterns:

- Context
- Problem/forces
- Solution

Properties of patterns:

- Addresses a recurring design problem.
- Identify and specify higher level abstractions.
- Provide a common vocabulary.
- Document existing, well-proven experience.
- Support constructing solutions.
- Help you manage complexity.

Advantages of (mature) patterns

- Codification:
 - In other fields of engineering: handbooks, prefabricated components (COTS).
 - In computing: component-palettes, libraries, handbooks.
- Common terminology: allows engineers to interact with one another.
- Well-understood interactions: internally, externally

Components: what do we want of them?

- Composibility.
- High performance (this is a *niche*).
- Well defined specification: substitutability.

Our approach

(Our template for analyzing patterns)

1. Give example occurrences of a pattern. Aim for several different applications.
2. Commonality analysis. What does each occurrence share.
3. Variability analysis. What are essential differences between occurrences.
4. Binding time/mechanisms. What are the mechanisms used to bind-in the differences; at what time are the bindings set:
 - design-in time;
 - instantiation time;
 - usage time, . . .

5. Formalisms. What underlies it all.

Revisiting cars and planes.

Idioms

Computing-related fields (mostly EE)

Transistors

- Totally new technology at the time. (One of the few true leaps — replacing tubes.)
- Several different technologies with varying physics/physical characteristics.
- Formalisms: simulation of physical aspects; rules-of-thumb.
- Production: (rudimentary) design palette.

Gates and standard cells

- Transistors often organized for logical functions: *gates*; specified by truth tables.
- Commonalities: (mostly) two input/single output (significant variations are possible).
- Variabilities/bindings:
 - number of inputs: cell design time.
 - truth table: cell design time, or final circuit production time (PGA), or installation time (FPGA).
- Formalisms: transformations between gate types (de Morgan); optimization methods (Karnaugh maps), design-assistance tools.
- Production: design palette (standard cell).

Flip-Flops

- Structural and behavioural patterns emerged.
- Several types: D, J-K, T, S-R, . . .
- Gate-level still required.
- Commonalities: two input/two output.
- Variabilities/bindings:
 - behavioural sequence: FF design time; PGA and FPGA also possible.
- Formalisms: manipulation and simplification techniques and models, design tools.
- Production: design palette.

Standard TTL-chip series

- Nothing really new.
- Production: codified and provided standard components:
 - 7400: NAND (4).
 - 7402: NOR (4).
 - 7474: J-K (2).
- Commonalities: packaging.
- Variabilities/bindings:
 - truth table: IC design time.
 - voltage: IC design time.
 - resilience: production/QA time.

- Formalisms: virtually unchanged.
- Production: dramatically simplified; design tools linked to CAM/costing.

ALUs

- Full computational units.
- Operations in arithmetic and bit-logical terms.
- Commonalities: broad set of operations.
- Variabilities/bindings:
 - specific operations: design time.
 - timing: design time.
 - interface: design time.
- Formalisms: introduction of instruction sets.
- Production: abstraction level also raised.

Rudimentary aspects of computing

16-bit multiplication

- Not supported on most 8-bit microprocessors.
- Simple code sequence.
- Repetition is ideal for assembler macros.
- Commonalities: fundamental operation being performed.
- Variabilities/bindings:
 - result location: assembly time.
 - registers as scratch: assembly time (could be earlier!).
 - operand values: run time.

- operand locations: assembly time
- Formalisms: simple pre/post (attention to side-effects).
- Codifications:
 - Macro libraries.
 - Leads to design of new instruction set.

Procedure calls and frames

- Factoring of code saves space.
- Register usage conventions introduced.
- This cannot always work (recursion).
- Commonalities: separated code.
- Variabilities/bindings:
 - separated code: assembly time, or run time.
 - arguments: run time.
 - argument locations: convention time, assembly time.
 - local variable structure: assembly time.

- local variable values: run time.
- Formalisms: more complex pre/post.
- Codifications:
 - Macros.
 - New instructions.

Microarchitectures

- Common instructions can be factored into a new instruction set.
- Introduce microarchitecture/programming.
- Commonalities: core set of instructions.
- Variabilities/bindings:
 - set of available instructions: architecture design time, or user run time.
 - program code: user run time.
- Formalism: (in principle) some semantic mapping between the layers.
- Production: made compiler writer's lives easier.

- Observation: this is very common:
 - VAXen.
 - Transmeta/Crusoe.

Assembly/programming language idioms

Alternation ('if-else')

- Decisions and alternative handling are common.
- Several structures are possible.
- Commonalities: notion of conditional execution.
- Variabilities/bindings:
 - control expression: compile-time (structure), run-time (value).
 - true code: compile-time.
 - false branch: compile-time.
 - code layout: compile-time.

- Formalism: Hoare logic, wp calculus, etc.
- Production:
 - reworked into macros,
 - migrated into programming languages,
 - some automated reasoning possible,

Repetitions ('loops')

- Some form of iteration is common.
- Commonalities: notions of control expression, loop body.
- Variabilities/bindings:
 - control expression: compile-time (structure), run-time (value).
 - body code: compile-time.
 - top/bottom checking: language-design time.
 - code layout: compile-time.
- Formalism: Hoare logic, wp calculus, etc.
- Production: as before.

Procedures and functions revisited

- Same reasoning as in assembly language.
- Commonalities: separated code.
- Variabilities/bindings:
 - separated code: compile time, or run time (procedure and function pointers).
 - arguments: run time.
 - argument locations: convention time, compile time.
 - local variable structure: compile time.
 - local variable values: run time.
- Formalisms: more complex pre/post.

- Codifications:
 - Procedure/function signature declaration.
 - Body definition.

Intra-language migrations

Arrays

- Heterogeneous sequences become common.
- Multiple dimensions possible.
- Bounds checking is common sense.
- Commonalities: heterogeneity.
- Variabilities/bindings:
 - element types: from compile time to run time (!!!).
 - number of dimensions: (originally run time possible) compile time.

- dimension sizes: from compile time (native C/C++) to run time (dope vectors).
 - bounds checking: language design, compile time (depends on compiler), link time.
 - memory layout: (broadly) compile time, but also depends on number/size of dimensions.
-
- Formalism: sequences already well understood.
 - Production: codified directly into languages, implemented in compilers.

Records and structures

- Heterogeneous tupling (arbitrary types, length) becomes common.
- Commonalities: top-level structure of each element of the tuple.
- Variabilities/bindings:
 - slice types: compile time.
 - number of slices: compile time.
 - sizes of slices: compile time, to run time.
 - memory layout: compile time.
- Formalism: tuples already well understood.
- Production: codified directly into languages as records, structures, or objects.

Strings

- Text manipulation makes this common.
- Commonalities: —
- Variabilities/bindings:
 - length: run time; queries can be memoized.
 - alphabet: language design time (?).
 - layout: language design time.
- Formalism: follows from arrays.
- Production: codified into languages — at various levels, cf. Awk, SNOBOL, versus C.

String copying

- Simple loops used — also deal with overlapping strings.
- Commonalities:
- Variabilities/bindings:
 - source and destination: compile time, to run time.
 - source length: run time.
 - destination sufficiency: pre-invocation, or on-the-fly.
- Formalism: follows from strings.
- Codification: hand-coded loops, C/C++ idioms, libraries, compiler intrinsics, language features (not necessarily in that order).