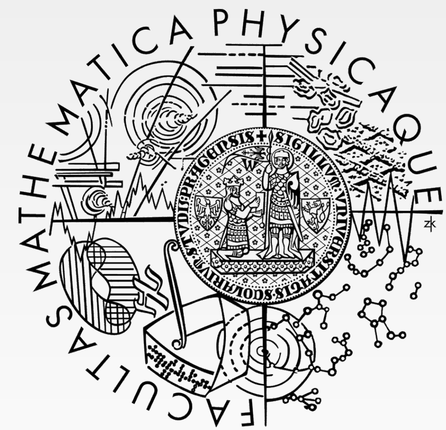
SOFA 2 overview

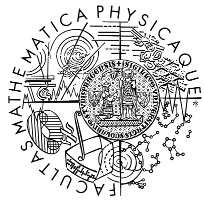
Petr Hnětynka

Charles University in Prague
Faculty of Mathematics and Physics

Czech Republic

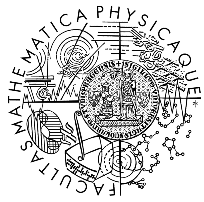


Overview



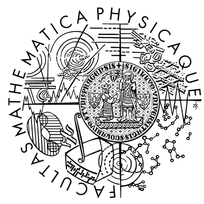
- SOFA 2 – a component system with hierarchical component model
- Based on SOFA (1)
  - features of the original SOFA
    - composite components
    - connectors (generated)
      - multiple communication styles
      - distributed deployment
    - versioning
    - behavior specification/verification
    - dynamic update
    - component trading/licensing
  - implemented in Java
    - freely available (LGPL)

SOFA 2 history



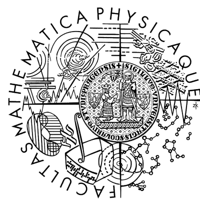
- Problems with SOFA (1)
  - evolution over years -> inconsistencies between implementation and specifications
    - e.g., protocols vs. connectors, architectures vs. dynamic reconfiguration,...
- SOFA 2 (2006)
  - properly balanced features
  - key improvements
    - based on meta-model
    - dynamic reconfiguration
    - explicitly modeled control part of components
    - support for multiple communication styles

SOFA 2 description outline



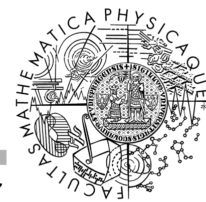
- Component model
  - meta-model
  - dynamic reconfiguration (dynamic architectures)
  - connectors
  - control parts (non-functional)
  - versioning
  - behavior specification
- Implementation
  - component lifecycle
  - runtime environment
  - usage, tools, current status

SOFA 2 definition

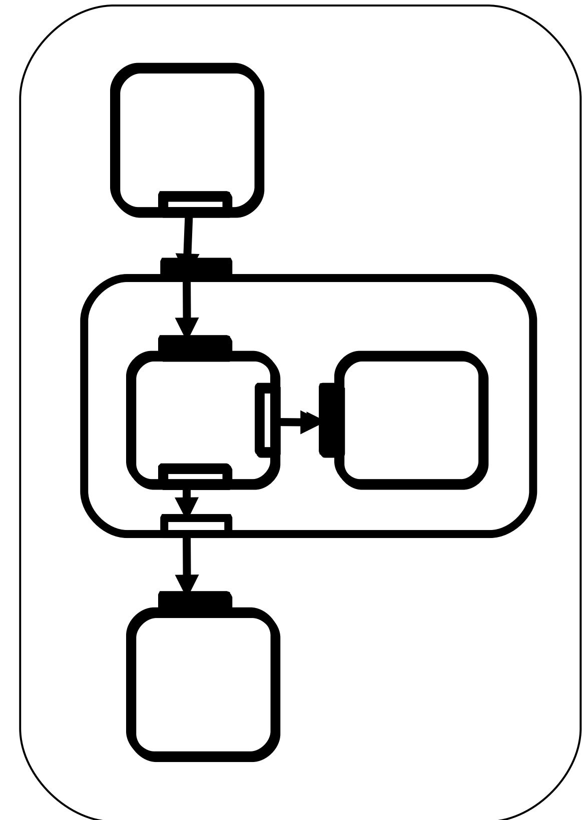


- Original SOFA
  - based on ADL
    - IDL-like syntax
    - added constructs for describing components
      - frames, architectures
- SOFA 2
  - based on meta-model (using EMF)
  - advantages
    - faster development
    - automated generation of a repository
    - existence of generators of models-editors
    - ...

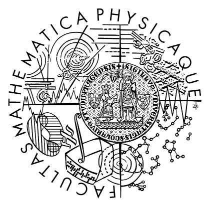
SOFA 2 components



- 2 basic abstractions
  - frame & architecture
- Frame
  - black-box view of a component
  - defines component's provided and required interfaces
    - interfaces defined by interface types
  - defines component's behavior
- Architecture
  - glass-box view
  - either primitive or composite
    - directly implemented or composed of other components
      - subcomponents defined primarily by frame



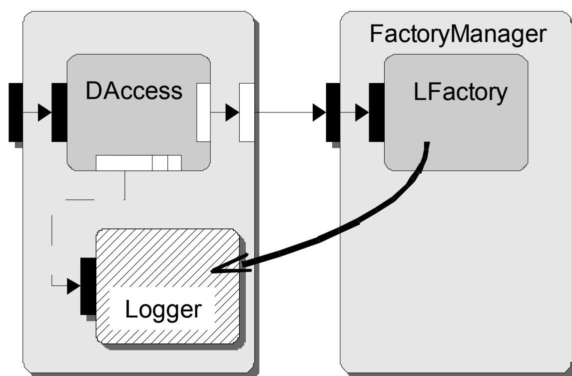
SOFA 2 meta-model



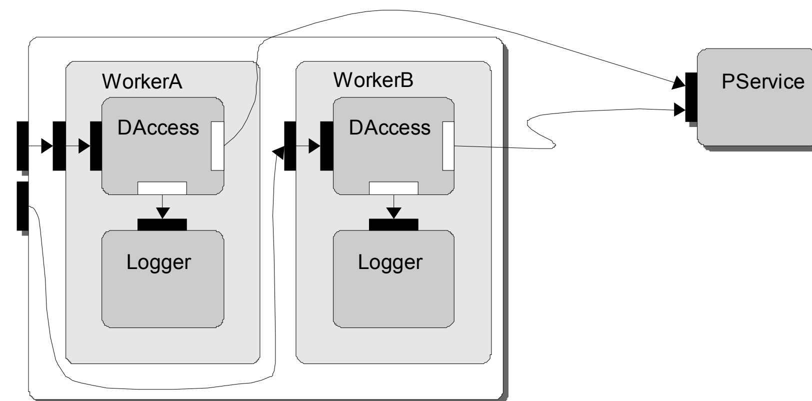
- Meta-model figure...

SOFA 2 dynamic architectures

- Support of dynamic architectures
  - i.e. changes of the architecture at runtime
  - via reconfiguration patterns
    - factory pattern (adding new components)
    - removal pattern
    - utility interface pattern (access to external services)



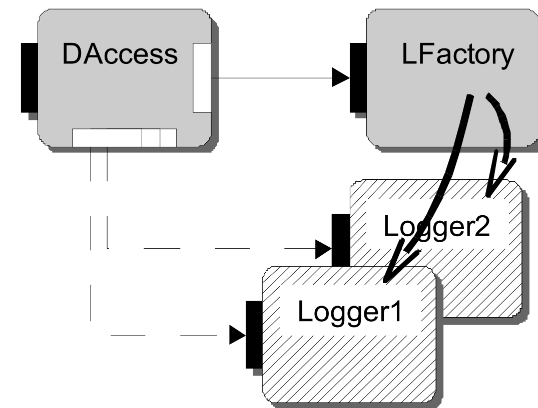
factory pattern



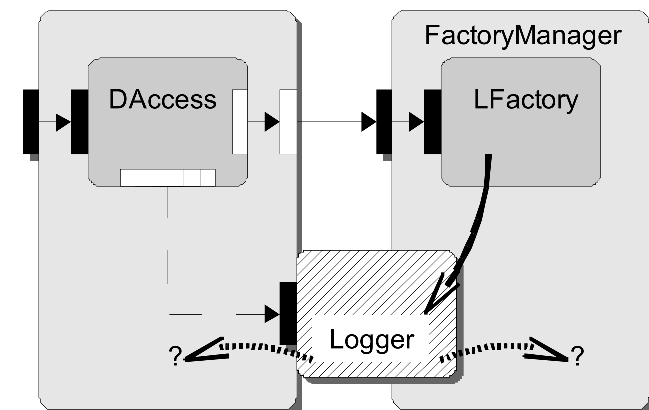
utility interface pattern

SOFA 2 dynamic architectures

- Why these patterns?
- Dynamic behavior is inherent to systems
  - simple example – multiple instances of a component
    - e.g. of parameterized loggers

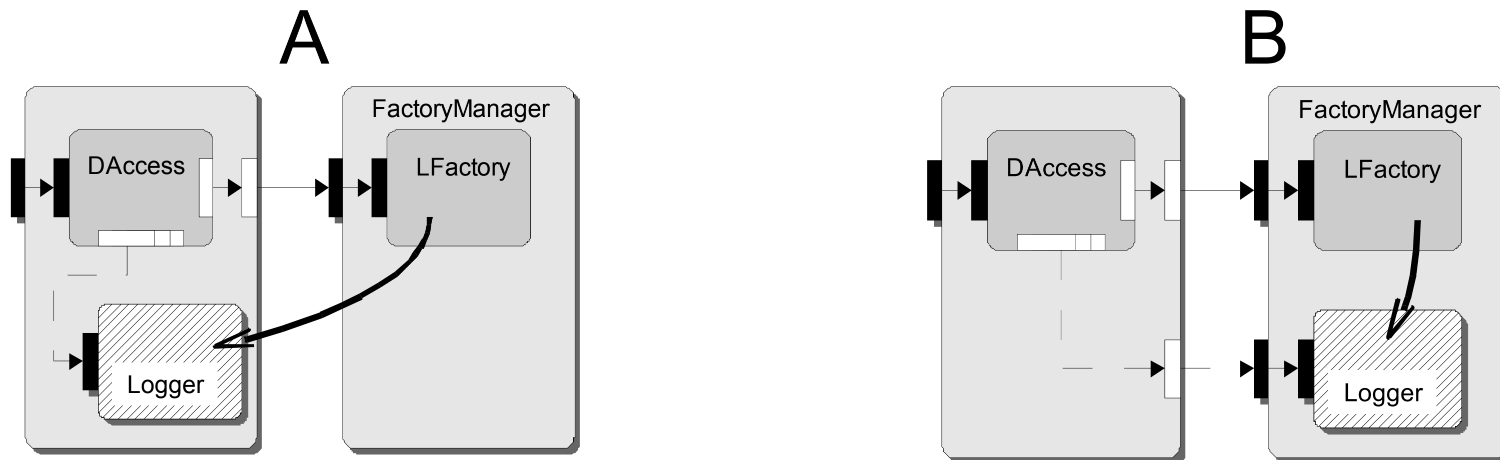


- In a flat component model
  - easy
- In a hierarchical component model
  - how to manage new components?



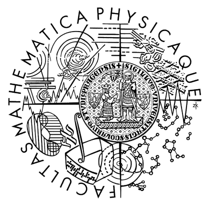
Nested factory pattern

- Two possibilities



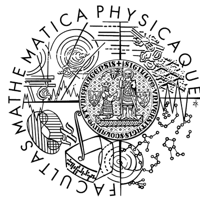
- We adopted A
  - component, which initiated the creation typically needs to intensively collaborate with the new component
  - B breaks the rule of well-defined component interface

Removal pattern



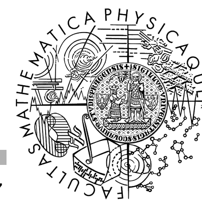
- Complementary to the factory pattern

Utility interfaces

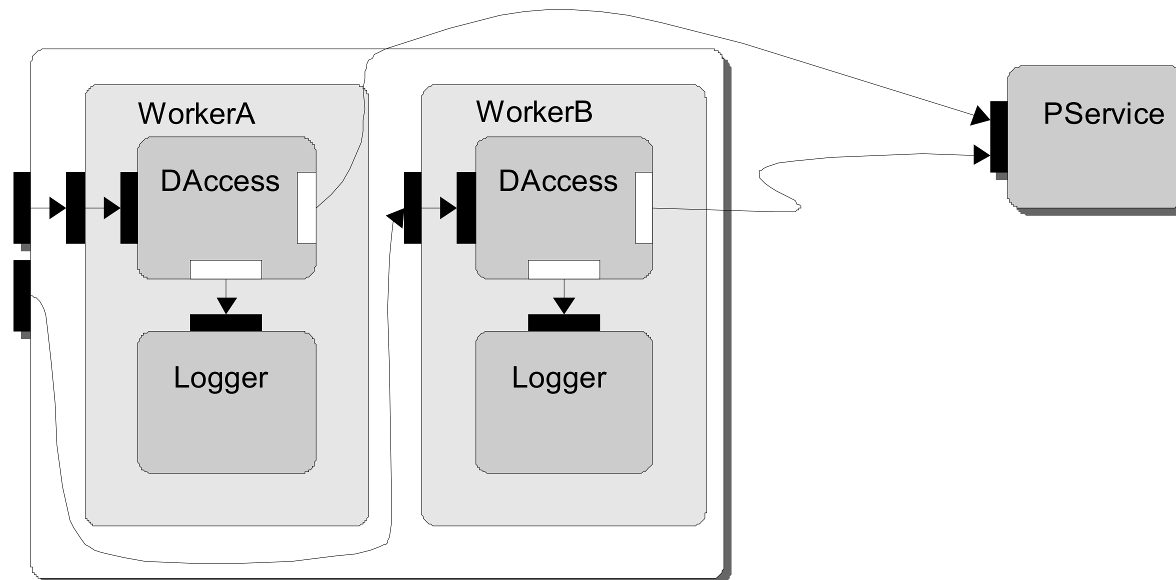


- Previous two patterns
  - nothing new
  - restricted reconfiguration
- Do not solve access to broadly-needed services
  - services needed by any component
  - strictly "component-based" solution – a component on the top level of the architecture hierarchy and connections through all the higher-level composite components
    - escalation of connections, unclean applications, performance penalties

Utility interfaces

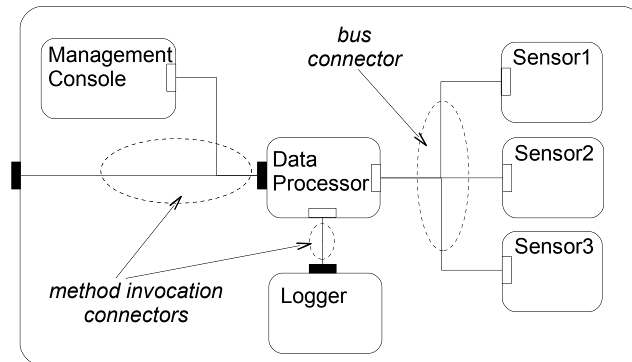


- New concept – a *utility* interface
  - reference to a utility interface can be freely passed among components
  - connection made to it established orthogonally to the architecture hierarchy



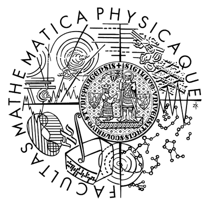
SOFA 2 connectors

- Connectors
  - connections among components
  - at design time
    - links with properties and communication style
      - method invocations, shared memory, messaging,...



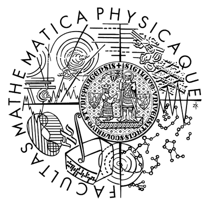
  - at deployment time
    - automatically generated based on properties and connected interfaces
    - allow transparently distributed applications

SOFA 2 control parts



- Explicitly separated control and business parts of components
- Control part
  - controls non-functional properties
  - provides so-called controllers
    - interfaces managing non-functional properties
  - modular and fully extensible
    - composed of microcomponents
    - applied as aspects at deployment time

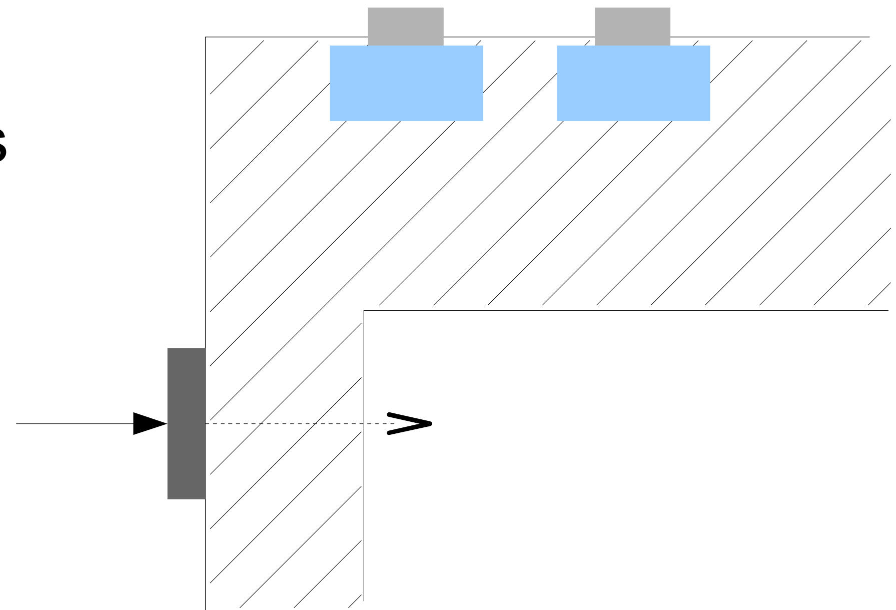
Microcomponent model



- Very minimalistic
- Flat
- No connectors
- No distribution
- No control part
- From the implementation view
  - microcomponent ~ class

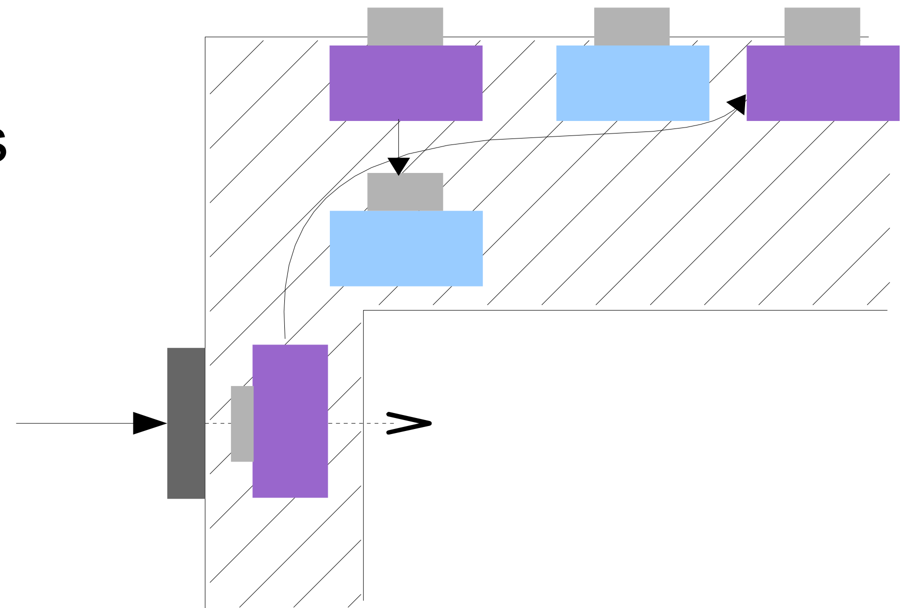
Aspects

- Defined on the top of the microcomponent model
- Aspect ~ extension of the control part
  - definition of microcomponents
  - instantiation patterns
- Core aspect
  - present in all components
  - controllers
    - lifecycle
    - binding

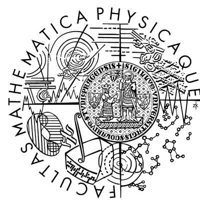


Aspects

- Defined on the top of the microcomponent model
- Aspect ~ extension of the control part
  - definition of microcomponents
  - instantiation patterns
- Core aspect
  - present in all components
  - controllers
    - lifecycle
    - binding



SOFA 2 component lifecycle

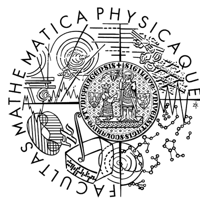


1. Development
   - composing existing components together
     - components stored in the repository
   - newly developed ones also stored in the repository
2. Assembly
   - subcomponents primarily defined by *frames*
   - recursively replacing *frames* by *architectures*
3. Deployment and executing
   - where particular components have to be executed
     - information stored in a *deployment plan*
   - connector generation
   - applying control aspects
   - execution

SOFA 2 Runtime environment

- Implementation in Java
- Runtime environment (called SOFAnode) consists of several units
  - **repository**
  - **deployment docks**
  - dock registry
  - global connector manager

SOFA 2 Runtime environment



- Repository
  - generated from the EMF meta-model
  - remotely accessible
  - adding new content via *cloning*
    - a developer creates new clone of the repository,
adds new content,
tests it and
finally merges it back
    - a clone can contain temporarily inconsistent content
    - at merge time, the content have to be consistent

SOFA 2 Runtime environment

- Deployment dock
  - a container executing components
  - automatically downloads code of components from the repository

Deployment Node 1

Repository

DockA

DockB

Deployment Node 2

Uploading new components

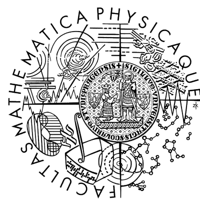
Developing new components

DockD

DockC

Deployment Node 3

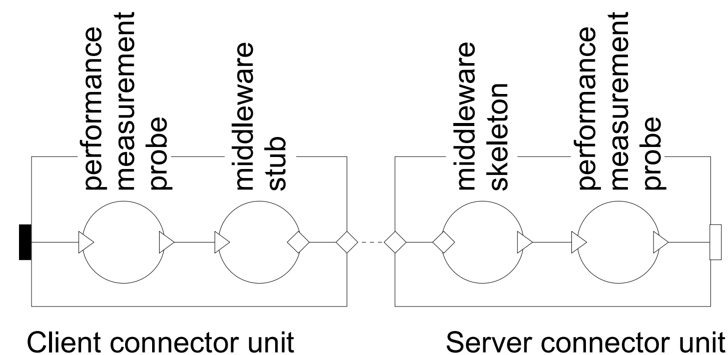
Component implementation



- Developers provide implementation just for primitive components
  - (composite components do not contain business code)
- Implementation
  - plain Java classes
  - no special requirements
    - about a number of threads, used libraries,...
  - currently
    - classes have to implement SOFA-specific interfaces
  - near future
    - provided and required services and initialization methods are marked by Java annotations
=> no external dependencies, code is reusable for different component system

Connectors

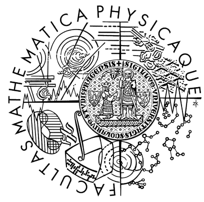
- Generated at deployment time
  - from the specification
    - communication style & properties (secure connection, logging...)
- Allows transparently distributed applications
  - developers do not bother with the networking
  - generated using suitable middleware
- Internal structure
  - set of connector elements
  - connector generator builds a suitable connector architecture and uses predefined elements
    - (connector architectures are either predefined or can be defined by special language)



Versioning

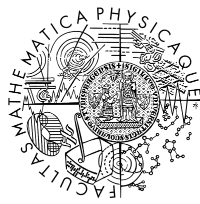
- SOFA 2 supports versioning of components
  - one component can exist in several versions
    - i.e. evolution of the component
  - versions are assigned by the repository
  - version identifiers are globally unique
- At runtime, it can lead to the *class name clashes* in Java virtual machine
  - i.e. a situation when a class/interface cannot be loaded to JVM because another class/interface with same name has been already loaded
  - it can happen e.g.
    - during dynamic update, or
    - single deployment dock hosts two applications – each of them use different version of the same component

Class name clashes



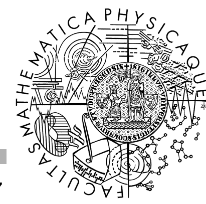
- Common solution
  - loading different components by different Java classloaders
    - Java identifies classes not only by the name but also by loading classloader
  - but it does not cover all sources of class name clashes
- Our solution
  - byte code manipulation
  - during uploading components to the repository, the names of classes are augmented in their byte code => unique names
  - completely transparent to the developers/users

Current status



- Implementation freely available (LGPL license)
  - http://sofa.objectweb.org/
- All features are implemented
  - “small issue”
    - not very “user-friendly”
  - we are working on graphical development tool
    - Eclipse based
- Cushion
  - command line development tool for SOFA 2 components

Development with Cushion



- Developing new application
  - create or reuse interface types, frames and architectures
    - `cushion new [interface,frame,architecture] ....`
    - `cushion commit ....`
    - `cushion checkout ....`
  - compiling Java code
    - `cushion compile ....`
    - `cushion upload ....`
  - assembling complete application
    - `cushion assembly`
  - deploying application
    - `cushion deplplan`
    - `cushion deploy`