Co-op: a case for custom, composable composition operators

Wilke Havinga, Christoph Bockisch and Lodewijk Bergmans

UNIVERSITY OF TWENTE.

March 15, 2010

Background

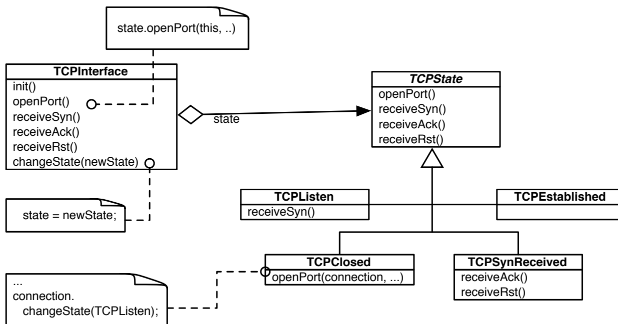
- Goal/trend: striving towards reusable high(er)-level abstractions in programming
- Example: at the *design level*, design patterns describe *reusable* abstractions addressing a recurring design problem
- A design pattern specifies a template for composition (structural, behavioral), *fixing* e.g. parts of structure or interaction patterns, but leaving everything else *variable*.

Example: GoF `State` Pattern

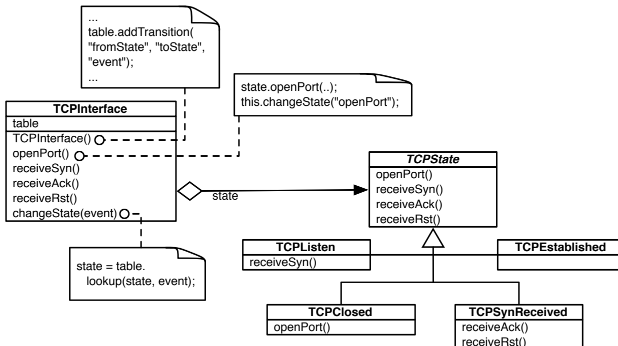
The GoF State pattern has the following “fixed” properties:

- Interface object where the external world sends all requests, thus encapsulating the pattern implementation
- Interface forwards requests to the active state (from a set of state objects), each implementing the desired behavioral specifications for a specific state
- Supports a mechanism by which the active state can be changed.
  - Var. 1: behavioral specifications within the state objects may ask the interface to switch to a new state
  - Var. 2: the interface contains a table that encodes state transitions, e.g. mapping current state + event to a new state

Example: `State` pattern instance, variant 1



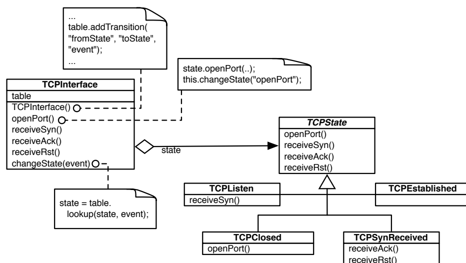
Example: `State` pattern instance, variant 2



Reusable abstractions?

- At the *design level*, a pattern is a reusable abstraction: the template can be applied to many instances of the problem it addresses.
- At the *implementation level*, can everything *fixed* by the pattern description be implemented as a reusable abstraction?

Reusable abstractions? (cont.)



- Everything that is fixed by the pattern is instantiated in a pattern-*instance*-specific way
- The fixed pattern-related code consists largely of repetitive boilerplate code (call forwarding, state change implementation, context parameter exchange)
- Fixed code parts not reusable, boilerplate has to be written for each pattern instance, whenever adding a new action, etc.

“You’re using the wrong programming language”

- Implementing (manual) forwarding of calls is unnecessary if your language supports explicit delegation
- The table-based state-change variant benefits from the use of aspects (after executing an action, update to new state)
- ...but do you know of (m)any languages that support both?

Composition & Variability in Programming Languages

Composition

- Programming languages support diverse composition operators
- E.g.: function calls, inheritance, delegation, pointcut-advice, composition filters, mixins, ...

Variability

- Many variants exist for each kind of composition operator
- E.g.: single vs. multiple inheritance, Beta vs. Smalltalk inheritance, ...

Motivation

Impact of programming language variability on software qualities

- The use of particular composition operators influences software quality factors
- E.g.: trade-offs between complexity, conciseness, reusability, ..
- Claim: no “silver bullet” composition operator exists, as trade-offs depend on the (application) domain, problem.

Problem: limited variability points within programming languages

- Typically, programming languages support a **fixed** set of composition operators, with **fixed** semantics.
- Forces programmers to “work around” language limitations
  - No suitable composition operator is supported
  - Composition operator semantics are not as desired
- Result: boilerplate (less concise code), decreased reusability

Approach: *Co-op*

Co-op is an experimental composition infrastructure that:

- Allows users to flexibly define composition operators (including many known OO, AO mechanisms) with variable semantics, using a small set of composition primitives
- Allows "arbitrary" combinations of composition operators within the same program
- Supports constraints to address interactions between operators
- Allows compositions of composition operators

Composition model

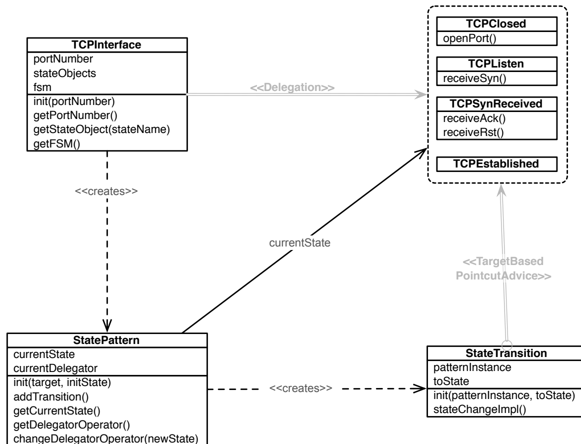
Composition Operators are constructed using a small set of primitive elements:

- *Events*, generated during execution of a program. Event properties: sender, (intended) target, name, annotation, parameters.
- *Event selectors*, may match events based on event properties and other reflective information
- *Action selectors*, select an operation to be invoked
- *Bindings* between event and action selectors, as well as between context variables on event- and invoked side
- *Constraints* between bindings (as event selectors of multiple bindings may match the same event)

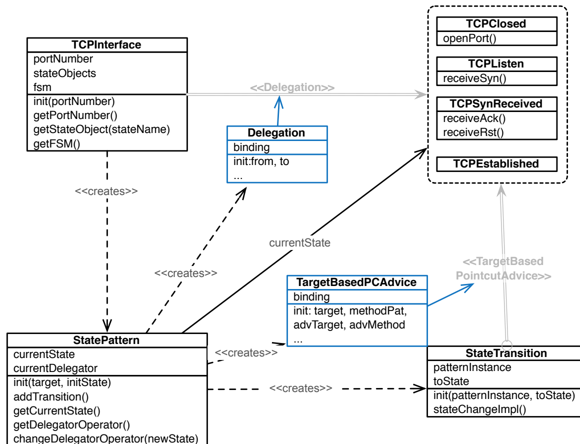
Prototype implementation

- To experiment with freely customizable composition operators, we defined an object-based language that purposely comes with only very rudimentary *built-in* composition support
- Within the language, users can modularly specify composition operators using the available composition primitives.
- The composition primitives are modeled as first-class entities (i.e., can be manipulated like normal objects), thus: can be parameterized, reused, refined, etc.

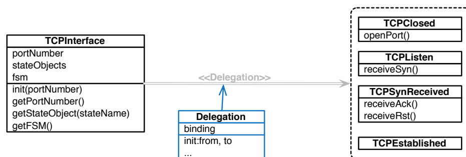
State pattern design using *Co-op*



State pattern design using *Co-op*



Delegation implementation



Delegation: *from*, *to*

- Event selector: events of which the target object is the object indicated by *from*
- Action selector: resend event with modified target object *to*.
- Context binding: bind "this" on the receiving side to "*from*"
- Constraint: Skip this binding if the "default dispatch" binding already evaluates successfully.

Target-based pointcut-advice implementation

TargetBasedPointcutAdvice: *target*, *methodPattern*, *adviceTarget*, *adviceMethod*

- Event selector: events with matching *target* and methodPattern properties
- Action selector: resend event with modified target object *adviceTarget* and message selector *adviceMethod*.
- Context binding: bind "this" on the receiving side to "target"
- Constraint: Evaluate the default binding *before* this binding, or vice versa (before/after advice)

Conclusion/Discussion

- *Co-op* allows expression of diverse composition operators using the same (small) set of primitives
  - Shown here for delegation, simple target-based pointcut-advice.
  - We have implemented many more, including several styles of inheritance (Smalltalk, Beta-style, multiple inheritance)
- Support for flexible, user-definable composition operators can improve modularity and reusability of design pattern implementations
  - Shown here for `State` pattern, we plan to apply this to other design patterns (observer, decorator, strategy, ...)
- Challenge: constraints can be used to deal with interactions between composition mechanisms, but interaction/interference analysis needs to be done manually